Raúl Rojas

Neural Networks

A Systematic Introduction

Springer
Berlin Heidelberg New York
Hong Kong London
Milan Paris Tokyo

Foreword

One of the well-springs of mathematical inspiration has been the continuing attempt to formalize human thought. From the syllogisms of the Greeks, through all of logic and probability theory, cognitive models have led to beautiful mathematics and wide ranging application. But mental processes have proven to be more complex than any of the formal theories and the various idealizations have broken off to become separate fields of study and application.

It now appears that the same thing is happening with the recent developments in connectionist and neural computation. Starting in the 1940s and with great acceleration since the 1980s, there has been an effort to model cognition using formalisms based on increasingly sophisticated models of the physiology of neurons. Some branches of this work continue to focus on biological and psychological theory, but as in the past, the formalisms are taking on a mathematical and application life of their own. Several varieties of adaptive networks have proven to be practical in large difficult applied problems and this has led to interest in their mathematical and computational properties.

We are now beginning to see good textbooks for introducing the subject to various student groups. This book by Raúl Rojas is aimed at advanced undergraduates in computer science and mathematics. This is a revised version of his German text which has been quite successful. It is also a valuable self-instruction source for professionals interested in the relation of neural network ideas to theoretical computer science and articulating disciplines.

The book is divided into eighteen chapters, each designed to be taught in about one week. The first eight chapters follow a progression and the later ones can be covered in a variety of orders. The emphasis throughout is on explicating the computational nature of the structures and processes and relating them to other computational formalisms. Proofs are rigorous, but not overly formal, and there is extensive use of geometric intuition and diagrams. Specific applications are discussed, with the emphasis on computational rather than engineering issues. There is a modest number of exercises at the end of most chapters.

The most widely applied mechanisms involve adapting weights in feed-forward networks of uniform differentiable units and these are covered thoroughly. In addition to chapters on the background, fundamentals, and variations on backpropagation techniques, there is treatment of related questions from statistics and computational complexity.

There are also several chapters covering recurrent networks including the general associative net and the models of Hopfield and Kohonen. Stochastic variants are presented and linked to statistical physics and Boltzmann learning. Other chapters (weeks) are dedicated to fuzzy logic, modular neural networks, genetic algorithms, and an overview of computer hardware developed for neural computation. Each of the later chapters is self-contained and should be readable by a student who has mastered the first half of the book.

The most remarkable aspect of neural computation at the present is the speed at which it is maturing and becoming integrated with traditional disciplines. This book is both an indication of this trend and a vehicle for bringing it to a generation of mathematically inclined students.

Berkeley, California Jerome Feldman

Preface

This book arose from my lectures on neural networks at the Free University of Berlin and later at the University of Halle. I started writing a new text out of dissatisfaction with the literature available at the time. Most books on neural networks seemed to be chaotic collections of models and there was no clear unifying theoretical thread connecting them. The results of my efforts were published in German by Springer-Verlag under the title *Theorie der neuronalen Netze*. I tried in that book to put the accent on a systematic development of neural network theory and to stimulate the intuition of the reader by making use of many figures. Intuitive understanding fosters a more immediate grasp of the objects one studies, which stresses the concrete meaning of their relations. Since then some new books have appeared, which are more systematic and comprehensive than those previously available, but I think that there is still much room for improvement. The German edition has been quite successful and at the time of this writing it has gone through five printings in the space of three years.

However, this book is not a translation. I rewrote the text, added new sections, and deleted some others. The chapter on fast learning algorithms is completely new and some others have been adapted to deal with interesting additional topics. The book has been written for undergraduates, and the only mathematical tools needed are those which are learned during the first two years at university. The book offers enough material for a semester, although I do not normally go through all chapters. It is possible to omit some of them so as to spend more time on others. Some chapters from this book have been used successfully for university courses in Germany, Austria, and the United States.

The various branches of neural networks theory are all interrelated closely and quite often unexpectedly. Even so, because of the great diversity of the material treated, it was necessary to make each chapter more or less self-contained. There are a few minor repetitions but this renders each chapter understandable and interesting. There is considerable flexibility in the order of presentation for a course. Chapter 1 discusses the biological motivation of the whole enterprise. Chapters 2, 3, and 4 deal with the basics of threshold logic and should be considered as a unit. Chapter 5 introduces vector quantization and unsupervised learning. Chapter 6 gives a nice geometrical interpretation of perceptron learning. Those interested in stressing current applications of neural networks can skip Chapters 5 and 6 and go directly to the backpropagation algorithm (Chapter 7). I am especially proud of this chapter because it introduces backpropagation with minimal effort, using a graphical approach, yet the result is more general than the usual derivations of the algorithm in other books. I was rather surprised to see that *Neural Computation* published in 1996 a paper about what is essentially the method contained in my German book of 1993.

Those interested in statistics and complexity theory should review Chapters 9 and 10. Chapter 11 is an *intermezzo* and clarifies the relation between fuzzy logic and neural networks. Recurrent networks are handled in the three chapters, dealing respectively with associative memories, the Hopfield model, and Boltzmann machines. They should be also considered a unit. The book closes with a review of self-organization and evolutionary methods, followed by a short survey of currently available hardware for neural networks.

We are still struggling with neural network theory, trying to find a more systematic and comprehensive approach. Every chapter should convey to the reader an understanding of one small additional piece of the larger picture. I sometimes compare the current state of the theory with a big puzzle which we are all trying to put together. This explains the small puzzle pieces that the reader will find at the end of each chapter. Enough discussion – Let us start our journey into the fascinating world of artificial neural networks without further delay.

Errata and electronic information

This book has an Internet home page. Any errors reported by readers, new ideas, and suggested exercises can be downloaded from Berlin, Germany. The WWW link is: http://www.inf.fu-berlin.de/∼rojas/neural. The home page offers also some additional useful information about neural networks. You can send your comments by e-mail to rojas@inf.fu-berlin.de.

Acknowledgements

Many friends and colleagues have contributed to the quality of this book. The names of some of them are listed in the preface to the German edition of 1993. Phil Maher, Rosi Weinert-Knapp, and Gaye Rochow revised my original manuscript. Andrew J. Ross, English editor at Springer-Verlag in Heidelberg, took great care in degermanizing my linguistic constructions.

The book was written at three different institutions: The Free University of Berlin provided an ideal working environment during the first phase of writing. Vilim Vesligaj configured TeX so that it would accept Springer's style.

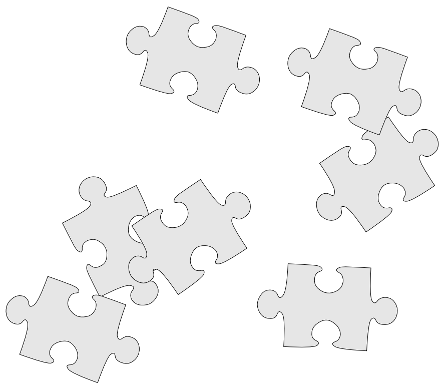
Günter Feuer, Marcus Pfister, Willi Wolf, and Birgit Müller were patient discussion partners. I had many discussions with Frank Darius on damned lies and statistics. The work was finished at Halle's Martin Luther University. My collaborator Bernhard Frötschl and some of my students found many of my early TeX-typos. I profited from two visits to the International Computer Science Institute in Berkeley during the summers of 1994 and 1995. I especially thank Jerry Feldman, Joachim Beer, and Nelson Morgan for their encouragement. Lokendra Shastri tested the backpropagation chapter "in the field", that is in his course on connectionist models at UC Berkeley. It was very rewarding to spend the evenings talking to Andres and Celina Albanese about other kinds of networks (namely real computer networks). Lotfi Zadeh was very kind in inviting me to present my visualization methods at his Seminar on Soft Computing. Due to the efforts of Dieter Ernst there is no good restaurant in the Bay Area where I have not been.

It has been a pleasure working with Springer-Verlag and the head of the planning section, Dr. Hans Wössner, in the development of this text. With him cheering from Heidelberg I could survive the whole ordeal of TeXing more than 500 pages.

Finally, I thank my daughter Tania and my wife Margarita Esponda for their love and support during the writing of this book. Since my German book was dedicated to Margarita, the new English edition is now dedicated to Tania. I really hope she will read this book in the future (and I hope she will like it).

Berlin and Halle
March 1996

Raúl Rojas González



“For Reason, in this sense, is nothing but
Reckoning (that is, Adding and Subtracting).”

Thomas Hobbes, *Leviathan.*

1

The Biological Paradigm

1.1 Neural computation

Research in the field of neural networks has been attracting increasing attention in recent years. Since 1943, when Warren McCulloch and Walter Pitts presented the first model of artificial neurons, new and more sophisticated proposals have been made from decade to decade. Mathematical analysis has solved some of the mysteries posed by the new models but has left many questions open for future investigations. Needless to say, the study of neurons, their interconnections, and their role as the brain's elementary building blocks is one of the most dynamic and important research fields in modern biology. We can illustrate the relevance of this endeavor by pointing out that between 1901 and 1991 approximately ten percent of the Nobel Prizes for Physiology and Medicine were awarded to scientists who contributed to the understanding of the brain. It is not an exaggeration to say that we have learned more about the nervous system in the last fifty years than ever before.

In this book we deal with *artificial neural networks*, and therefore the first question to be clarified is their relation to the biological paradigm. What do we abstract from real neurons for our models? What is the link between neurons and artificial computing units? This chapter gives a preliminary answer to these important questions.

1.1.1 Natural and artificial neural networks

Artificial neural networks are an attempt at modeling the information processing capabilities of nervous systems. Thus, first of all, we need to consider the essential properties of biological neural networks from the viewpoint of information processing. This will allow us to design abstract models of artificial neural networks, which can then be simulated and analyzed.

Although the models which have been proposed to explain the structure of the brain and the nervous systems of some animals are different in many respects, there is a general consensus that the essence of the operation of neural ensembles is “control through communication” [72]. Animal nervous systems are composed of thousands or millions of interconnected cells. Each one of them is a very complex arrangement which deals with incoming signals in many different ways. However, neurons are rather slow when compared to electronic logic gates. These can achieve switching times of a few nanoseconds, whereas neurons need several milliseconds to react to a stimulus. Nevertheless the brain is capable of solving problems which no digital computer can yet efficiently deal with.

Massive and hierarchical networking of the brain seems to be the fundamental precondition for the emergence of consciousness and complex behavior [202]. So far, however, biologists and neurologists have concentrated their research on uncovering the properties of individual neurons. Today, the mechanisms for the production and transport of signals from one neuron to the other are well-understood physiological phenomena, but how these individual systems cooperate to form complex and massively parallel systems capable of incredible information processing feats has not yet been completely elucidated. Mathematics, physics, and computer science can provide invaluable help in the study of these complex systems. It is not surprising that the study of the brain has become one of the most interdisciplinary areas of scientific research in recent years.

However, we should be careful with the metaphors and paradigms commonly introduced when dealing with the nervous system. It seems to be a constant in the history of science that the brain has always been compared to the most complicated contemporary artifact produced by human industry [297]. In ancient times the brain was compared to a pneumatic machine, in the Renaissance to a clockwork, and at the end of the last century to the telephone network. There are some today who consider computers the paradigm par excellence of a nervous system. It is rather paradoxical that when John von Neumann wrote his classical description of future universal computers, he tried to choose terms that would describe computers in terms of brains, not brains in terms of computers.

The nervous system of an animal is an information processing totality. The sensory inputs, i.e., signals from the environment, are coded and processed to evoke the appropriate response. Biological neural networks are just one of many possible solutions to the problem of processing information. The main difference between neural networks and conventional computer systems is the massive parallelism and redundancy which they exploit in order to deal with the unreliability of the individual computing units. Moreover, biological neural networks are self-organizing systems and each individual neuron is also a delicate self-organizing structure capable of processing information in many different ways.

In this book we study the information processing capabilities of complex hierarchical networks of simple computing units. We deal with systems whose structure is only partially predetermined. Some parameters modify the capabilities of the network and it is our task to find the best combination for the solution of a given problem. The adjustment of the parameters will be done through a *learning algorithm*, i.e., not through explicit programming but through an automatic adaptive method.

A cursory review of the relevant literature on artificial neural networks leaves the impression of a chaotic mixture of very different network topologies and learning algorithms. Commercial neural network simulators sometimes offer several dozens of possible models. The large number of proposals has led to a situation in which each single model appears as part of a big puzzle whereas the bigger picture is absent. Consequently, in the following chapters we try to solve this puzzle by systematically introducing and discussing each of the neural network models in relation to the others.

Our approach consists of stating and answering the following questions: what information processing capabilities emerge in hierarchical systems of primitive computing units? What can be computed with these networks? How can these networks determine their structure in a self-organizing manner?

We start by considering biological systems. Artificial neural networks have aroused so much interest in recent years, not only because they exhibit interesting properties, but also because they try to mirror the kind of information processing capabilities of nervous systems. Since information processing consists of transforming signals, we deal with the biological mechanisms for their generation and transmission in this chapter. We discuss those biological processes by which neurons produce signals, and absorb and modify them in order to retransmit the result. In this way biological neural networks give us a clue regarding the properties which would be interesting to include in our artificial networks.

1.1.2 Models of computation

Artificial neural networks can be considered as just another approach to the problem of computation. The first formal definitions of computability were proposed in the 1930s and '40s and at least five different alternatives were studied at the time. The computer era was started, not with one single approach, but with a contest of alternative computing models. We all know that the von Neumann computer emerged as the undisputed winner in this confrontation, but its triumph did not lead to the dismissal of the other computing models. Figure 1.1 shows the five principal contenders:

The mathematical model

Mathematicians avoided dealing with the problem of a function's computability until the beginning of this century. This happened not just because existence theorems were considered sufficient to deal with functions, but mainly because nobody had come up with a satisfactory definition of *computability*, certainly a relative concept which depends on the specific tools that can be used. The general solution for algebraic equations of degree five, for example, cannot be formulated using only algebraic functions, yet this can be done if a more general class of functions is allowed as computational primitives. The squaring of the circle, to give another example, is impossible using ruler and compass, but it has a trivial real solution.

If we want to talk about computability we must therefore specify which tools are available. We can start with the idea that some primitive functions and composition rules are "obviously" computable. All other functions which can be expressed in terms of these primitives and composition rules are then also computable.

David Hilbert, the famous German mathematician, was the first to state the conjecture that a certain class of functions contains all intuitively computable functions. Hilbert was referring to the primitive recursive functions, the class of functions which can be constructed from the zero and successor function using composition, projection, and a deterministic number of iterations (primitive recursion). However, in 1928, Wilhelm Ackermann was able to find a computable function which is not primitive recursive. This led to the definition of the general recursive functions [154]. In this formalism, a new composition rule has to be introduced, the so-called μ operator, which is equivalent to an indeterminate recursion or a lookup in an infinite table. At the same time Alonzo Church and collaborators developed the lambda calculus, another alternative to the mathematical definition of the computability concept [380]. In 1936, Church and Kleene were able to show that the general recursive functions can be expressed in the formalism of the lambda calculus. This led to the *Church thesis* that computable functions are the general recursive functions. David Deutsch has recently added that this thesis should be considered to be a statement about the physical world and be given the same status as a physical principle. He thus speaks of a "Church principle" [109].

The logic-operational model (Turing machines)

In his classical paper "On Computable Numbers with an Application to the Entscheidungsproblem" Alan Turing introduced another kind of computing model. The advantage of his approach is that it consists in an operational, mechanical model of computability. A Turing machine is composed of an infinite tape, in which symbols can be stored and read again. A read-write head can move to the left or to the right according to its internal state, which is updated at each step. The *Turing thesis* states that computable functions are those which can be computed with this kind of device. It was formulated concurrently with the Church thesis and Turing was able to show almost immediately that they are equivalent [435]. The Turing approach made clear for the first time what "programming" means, curiously enough at a time when no computer had yet been built.

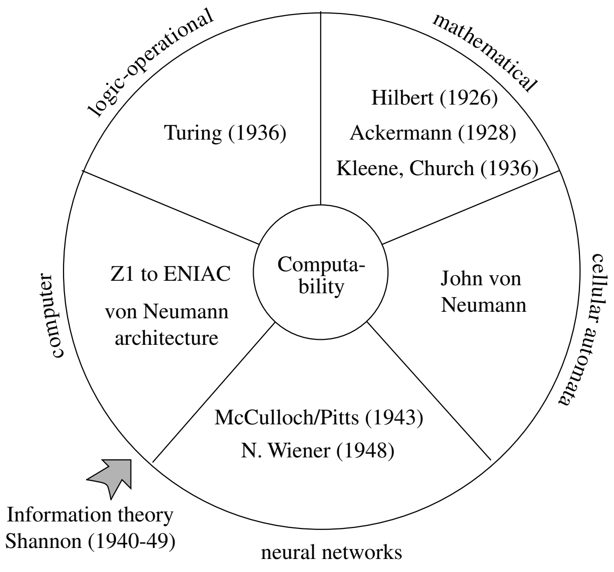


Fig. 1.1. Five models of computation

The computer model

The first electronic computing devices were developed in the 1930s and '40s. Since then, "computation-with-the-computer" has been regarded as computability itself. However the first engineers developing computers were for the most part unaware of Turing's or Church's research. Konrad Zuse, for example, developed in Berlin between 1938 and 1944 the computing machines Z1 and Z3 which were programmable but not universal, because they could not reach the whole space of the computable functions. Zuse's machines were able to process a sequence of instructions but could not iterate. Other computers of the time, like the Mark I built at Harvard, could iterate a constant number of times but were incapable of executing open-ended iterations (WHILE loops). Therefore the Mark I could compute the primitive but not the general recursive functions. Also the ENIAC, which is usually hailed as the world's first electronic computer, was incapable of dealing with open-ended loops, since iterations were determined by specific connections between modules of the machine. It seems that the first universal computer was the Mark I built in Manchester [96, 375]. This machine was able to cover all computable functions by making use of conditional branching and self-modifying programs, which is one possible way of implementing indexed addressing [268].

Cellular automata

The history of the development of the first mechanical and electronic computing devices shows how difficult it was to reach a consensus on the architecture of universal computers. Aspects such as the economy or the dependability of the building blocks played a role in the discussion, but the main problem was the definition of the minimal architecture needed for universality. In machines like the Mark I and the ENIAC there was no clear separation between memory and processor, and both functional elements were intertwined. Some machines still worked with base 10 and not 2, some were sequential and others parallel.

John von Neumann, who played a major role in defining the architecture of sequential machines, analyzed at that time a new computational model which he called *cellular automata*. Such automata operate in a "computing space" in which all data can be processed simultaneously. The main problem for cellular automata is communication and coordination between all the computing cells. This can be guaranteed through certain algorithms and conventions. It is not difficult to show that all computable functions, in the sense of Turing, can also be computed with cellular automata, even of the one-dimensional type, possessing only a few states. Turing himself considered this kind of computing model at one point in his career [192].

Cellular automata as computing model resemble massively parallel multiprocessor systems of the kind that has attracted considerable interest recently.

The biological model (neural networks)

The explanation of important aspects of the physiology of neurons set the stage for the formulation of artificial neural network models which do not operate sequentially, as Turing machines do. Neural networks have a hierarchical multilayered structure which sets them apart from cellular automata, so that information is transmitted not only to the immediate neighbors but also to more distant units. In artificial neural networks one can connect each unit to any other. In contrast to conventional computers, no program is handed over to the hardware – such a program has to be created, that is, the free parameters of the network have to be found adaptively.

Although neural networks and cellular automata are potentially more efficient than conventional computers in certain application areas, at the time of their conception they were not yet ready to take center stage. The necessary theory for harnessing the dynamics of complex parallel systems is still being developed right before our eyes. In the meantime, conventional computer technology has made great strides.

There is no better illustration for the simultaneous and related emergence of these various computability models than the life and work of John von Neumann himself. He participated in the definition and development of at least three of these models: in the architecture of sequential computers [417], the theory of cellular automata and the first neural network models. He also collaborated with Church and Turing in Princeton [192].

Artificial neural networks have, as initial motivation, the structure of biological systems, and constitute an alternative computability paradigm. For that reason we will review some aspects of the way in which biological systems perform information processing. The fascination which still pervades this research field has much to do with the points of contact with the surprisingly elegant methods used by neurons in order to process information at the cellular level. Several million years of evolution have led to very sophisticated solutions to the problem of dealing with an uncertain environment. In this chapter we will discuss some elements of these strategies in order to determine what features we want to adopt in our abstract models of neural networks.

1.1.3 Elements of a computing model

What are the *elementary components* of any conceivable computing model? In the theory of general recursive functions, for example, it is possible to reduce any computable function to some composition rules and a small set of primitive functions. For a universal computer, we ask about the existence of a minimal and sufficient instruction set. For an arbitrary computing model the following metaphoric expression has been proposed:

$$computation = storage + transmission + processing.$$

The mechanical computation of a function presupposes that these three elements are present, that is, that data can be stored, communicated to the functional units of the model and transformed. It is implicitly assumed that a certain coding of the data has been agreed upon. Coding plays an important role in information processing because, as Claude Shannon showed in 1948, when noise is present information can still be transmitted without loss, if the right code with the right amount of redundancy is chosen.

Modern computers transform storage of information into a form of information transmission. Static memory chips store a bit as a circulating current until the bit is read. Turing machines store information in an infinite tape, whereas transmission is performed by the read-write head. Cellular automata store information in each cell, which at the same time is a small processor.

1.2 Networks of neurons

In biological neural networks information is stored at the contact points between different neurons, the so-called *synapses*. Later we will discuss what role these elements play for the storage, transmission, and processing of information. Other forms of storage are also known, because neurons are themselves complex systems of self-organizing signaling. In the next few pages we cannot do justice to all this complexity, but we analyze the most salient features and, with the metaphoric expression given above in mind, we will ask: how do neurons compute?

1.2.1 Structure of the neurons

Nervous systems possess global architectures of variable complexity, but all are composed of similar building blocks, the neural cells or neurons. They can perform different functions, which in turn leads to a very variable morphology. If we analyze the human cortex under a microscope, we can find several different types of neurons. Figure 1.2 shows a diagram of a portion of the cortex. Although the neurons have very different forms, it is possible to recognize a hierarchical structure of six different layers. Each one has specific functional characteristics. Sensory signals, for example, are transmitted directly to the fourth layer and from there processing is taken over by other layers.

Fig. 1.2. A view of the human cortex [from Lassen et al. 1988]

Neurons receive signals and produce a response. The general structure of a generic neuron is shown in Figure 1.3[1]. The branches to the left are the transmission channels for incoming information and are called *dendrites*. Dendrites receive the signals at the contact regions with other cells, the synapses

[1] Some animals have neurons with a very different morphology. In insects, for example, the dendrites go directly into the axon and the cell body is located far from them. The way these neurons work is nevertheless very similar to the description in this chapter.

mentioned already. Organelles in the body of the cell produce all necessary chemicals for the continuous working of the neuron. The mitochondria, visible in Figure 1.3, can be thought of as part of the energy supply of the cell, since they produce chemicals which are consumed by other cell structures. The output signals are transmitted by the *axon*, of which each cell has at most one. Some cells do not have an axon, because their task is only to set some cells in contact with others (in the retina, for example).

Fig. 1.3. A typical motor neuron [from Stevens 1988]

These four elements, dendrites, synapses, cell body, and axon, are the minimal structure we will adopt from the biological model. Artificial neurons for computing will have input channels, a cell body and an output channel. Synapses will be simulated by contact points between the cell body and input or output connections; a *weight* will be associated with these points.

1.2.2 Transmission of information

The fundamental problem of any information processing system is the transmission of information, as data storage can be transformed into a recurrent transmission of information between two points [177].

Biologists have known for more than 100 years that neurons transmit information using electrical signals. Because we are dealing with biological structures, this cannot be done by simple electronic transport as in metallic cables. Evolution arrived at another solution involving ions and semipermeable membranes.

Our body consists mainly of water, 55% of which is contained within the cells and 45% forming its environment. The cells preserve their identity and biological components by enclosing the protoplasm in a membrane made of a double layer of molecules that form a diffusion barrier. Some salts, present in our body, dissolve in the intracellular and extracellular fluid and dissociate into negative and positive ions. Sodium chloride, for example, dissociates into positive sodium ions (Na^+) and negative chlorine ions (Cl^-). Other positive ions present in the interior or exterior of the cells are potassium (K^+) and calcium (Ca^{2+}). The membranes of the cells exhibit different degrees of permeability for each one of these ions. The permeability is determined by the number and size of pores in the membrane, the so-called *ionic channels*. These are macromolecules with forms and charges which allow only certain ions to go from one side of the cell membrane to the other. Channels are selectively permeable to sodium, potassium or calcium ions. The specific permeability of the membrane leads to different distributions of ions in the interior and the exterior of the cells and this, in turn, to the interior of neurons being negatively charged with respect to the extracellular fluid.

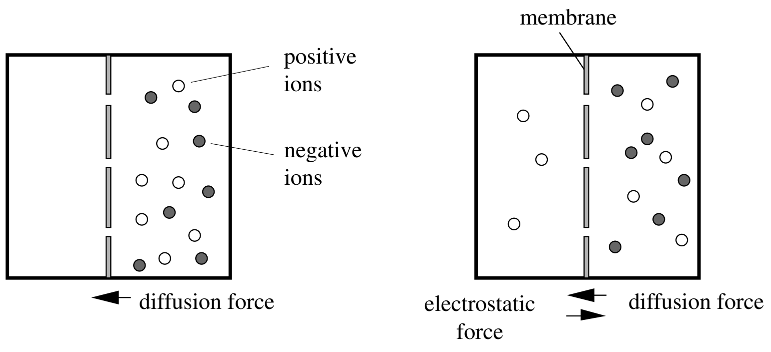


Fig. 1.4. Diffusion of ions through a membrane

Figure 1.4 illustrates this phenomenon. A box is divided into two parts separated by a membrane permeable only to positive ions. Initially the same number of positive and negative ions is located in the right side of the box. Later, some positive ions move from the right to the left through the pores in the membrane. This occurs because atoms and molecules have a thermodynamical tendency to distribute homogeneously in space by the process called diffusion. The process continues until the electrostatic repulsion from the positive ions on the left side balances the diffusion potential. A potential difference, called the *reversal potential*, is established and the system behaves like a small electric battery. In a cell, if the initial concentration of potassium ions in its interior is greater than in its exterior, positive potassium ions will diffuse through the open potassium-selective channels. If these are the only ionic channels, negative ions cannot disperse through the membrane. The interior of the cell becomes negatively charged with respect to the exterior, creating a potential difference between both sides of the membrane. This balances the diffusion potential, and, at some point, the net flow of potassium ions through the membrane falls to zero. The system reaches a steady state. The potential difference E for one kind of ion is given by the Nernst formula

$$E = k(\ln(c_o) - \ln(c_i))$$

where c_i is the concentration inside the cell, c_o the concentration in the extracellular fluid and k is a proportionality constant [295]. For potassium ions the equilibrium potential is -80 mV.

Because there are several different concentrations of ions inside and outside of the cell, the question is, what is the potential difference which is finally reached. The exact potential in the interior of the cell depends on the mixture of concentrations. A typical cell's potential is -70 mV, which is produced mainly by the ion concentrations shown in Figure 1.5 (A^- designates negatively charged biomolecules). The two main ions in the cell are sodium and potassium. Equilibrium potential for sodium lies around 58 mV. The cell reaches a potential between -80 mV and 58 mV. The cell's equilibrium potential is nearer to the value induced by potassium, because the permeability of the membrane to potassium is greater than to sodium. There is a net outflow of potassium ions at this potential and a net inflow of sodium ions. However, the sodium ions are less mobile because fewer open channels are available. In the steady state the cell membrane experiences two currents of ions trying to reach their individual equilibrium potential. An ion pump guarantees that the concentration of ions does not change with time.

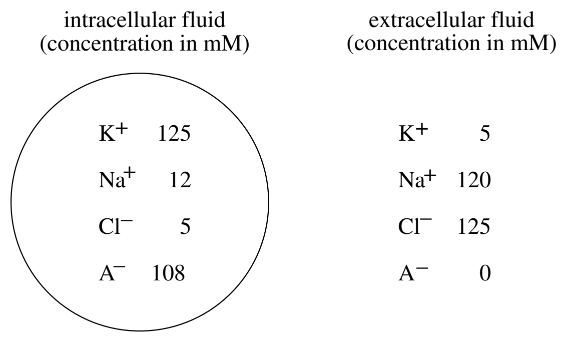


Fig. 1.5. Ion concentrations inside and outside a cell

The British scientists Alan Hodgkin and Andrew Huxley were able to show that it is possible to build an electric model of the cell membrane based on very simple assumptions. The membrane behaves as a capacitor made of two isolated layers of lipids. It can be charged with positive or negative ions. The different concentrations of several classes of ions in the interior and exterior of the cell provide an energy source capable of negatively polarizing the interior of the cell. Figure 1.6 shows a diagram of the model proposed by Hodgkin and

Huxley. The specific permeability of the membrane for each class of ion can be modeled like a conductance (the reciprocal of resistance).

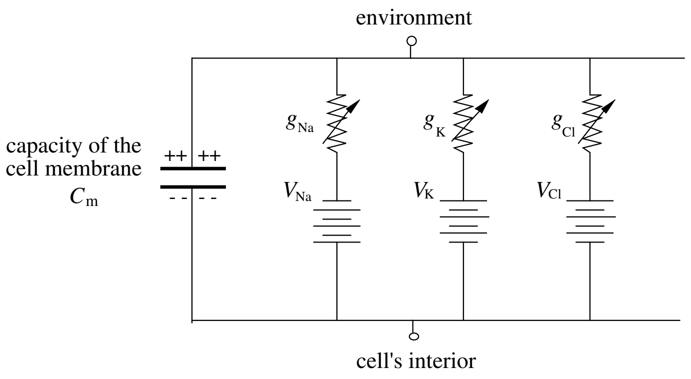


Fig. 1.6. The Hodgkin–Huxley model of a cell membrane

The electric model is a simplification, because there are other classes of ions and electrically charged proteins present in the cell. In the model, three ions compete to create a potential difference between the interior and exterior of the cell. The conductances g_{Na}, g_K, and g_L reflect the permeability of the membrane to sodium, potassium, and leakages, i.e., the number of open channels of each class. A signal can be produced by modifying the polarity of the cell through changes in the conductances g_{Na} and g_K. By making g_{Na} larger and the mobility of sodium ions greater than the mobility of potassium ions, the polarity of the cell changes from -70 mV to a positive value, nearer to the 58 mV at which sodium ions reach equilibrium. If the conductance g_K then becomes larger and g_{Na} falls back to its original value, the interior of the cell becomes negative again, overshooting in fact by going below -70 mV. To generate a signal, a mechanism for depolarizing and polarizing the cell in a controlled way is necessary.

The conductance and resistance of a cell membrane in relation to the different classes of ions depends on its permeability. This can be controlled by opening or closing excitable ionic channels. In addition to the static ionic channels already mentioned, there is another class which can be electrically controlled. These channels react to a depolarization of the cell membrane. When this happens, that is, when the potential of the interior of the cell in relation to the exterior reaches a threshold, the sodium-selective channels open automatically and positive sodium ions flow into the cell making its interior positive. This in turn leads to the opening of the potassium-selective channels and positive potassium ions flow to the exterior of the cell, restoring the original negative polarization.

Figure 1.7 shows a diagram of an electrically controlled sodium-selective channel which lets only sodium ions flow across. This effect is produced by the small aperture in the middle of the channel which is negatively charged (at time $t = 1$). If the interior of the cell becomes positive relative to the exterior, some negative charges are displaced in the channel and this produces the opening of a gate ($t = 2$). Sodium ions flow through the channel and into the cell. After a short time the second gate is closed and the ionic channel is sealed ($t = 3$). The opening of the channel corresponds to a change of membrane conductivity as explained above.

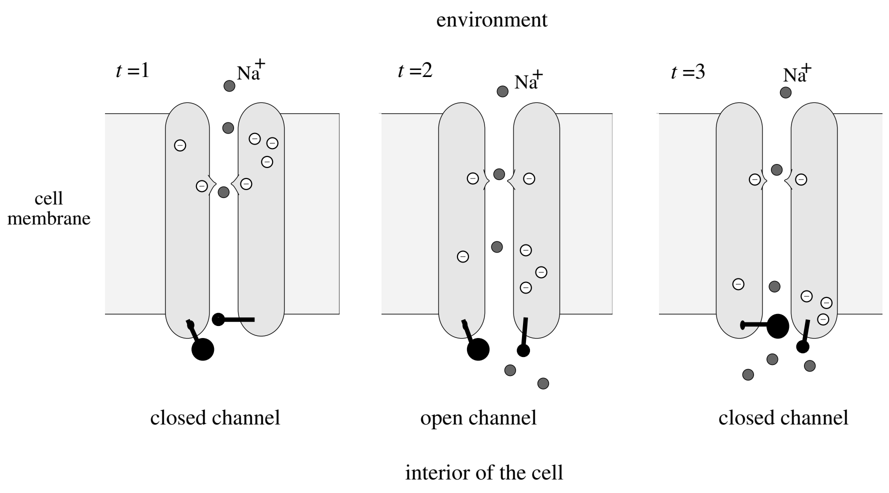


Fig. 1.7. Electrically controlled ionic channels

Static and electrically controlled ionic channels are not only found in neurons. As in any electrical system there are charge losses which have to be continuously balanced. A sodium ion pump (Figure 1.8) transports the excess of sodium ions out of the cell and, at the same time, potassium ions into its interior. The ion pump consumes adenosine triphosphate (ATP), a substance produced by the mitochondria, helping to stabilize the polarization potential of −70 mV. The ion pump is an example of a self-regulating system, because it is accelerated or decelerated by the differences in ion concentrations on both sides of the membrane. Ion pumps are constantly active and account for a considerable part of the energy requirements of the nervous system.

Neural signals are produced and transmitted at the cell membrane. The signals are represented by depolarization waves traveling through the axons in a self-regenerating manner. Figure 1.9 shows the form of such a depolarization wave, called an *action potential*. The x-dimension is shown horizontally and the diagram shows the instantaneous potential in each segment of the axon.

An action potential is produced by an initial depolarization of the cell membrane. The potential increases from −70 mV up to +40 mV. After some time the membrane potential becomes negative again but it overshoots, going

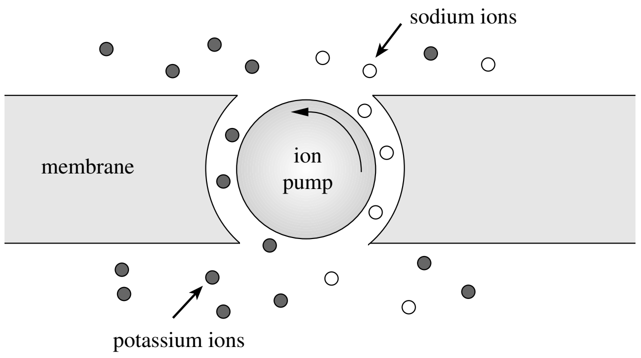


Fig. 1.8. Sodium and potassium ion pump

as low as −80 mV. The cell recovers gradually and the cell membrane returns to the initial potential. The switching time of the neurons is determined, as in any resistor-capacitor configuration, by the RC constant. In neurons, 2.4 milliseconds is a typical value for this constant.

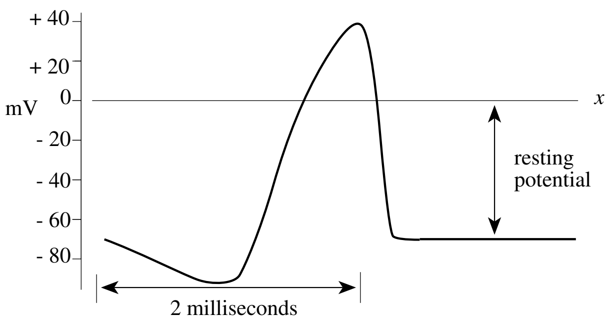


Fig. 1.9. Typical form of the action potential

Figure 1.10 shows an action potential traveling through an axon. A local perturbation, produced by the signals arriving at the dendrites, leads to the opening of the sodium-selective channels in a certain region of the cell membrane. The membrane is thus depolarized and positive sodium ions flow into the cell. After a short delay, the outward flow of potassium ions compensates the depolarization of the membrane. Both perturbations – the opening of the sodium and potassium-selective channels – are transmitted through the axon like falling dominos. In the entire process only local energy is consumed, that is, only the energy stored in the polarized membrane itself. The action potential is thus a wave of Na^+ permeability increase followed by a wave of K^+ permeability increase. It is easy to see that charged particles only move a short distance in the direction of the perturbation, only as much as is necessary to perturb the next channels and bring the next “domino” to fall.

Figure 1.10 also shows how impulse trains are produced in the cells. After a signal is produced a new one follows. Each neural signal is an all-or-nothing self-propagating regenerative event as each signal has the same form and amplitude. At this level we can safely speak about digital transmission of information.

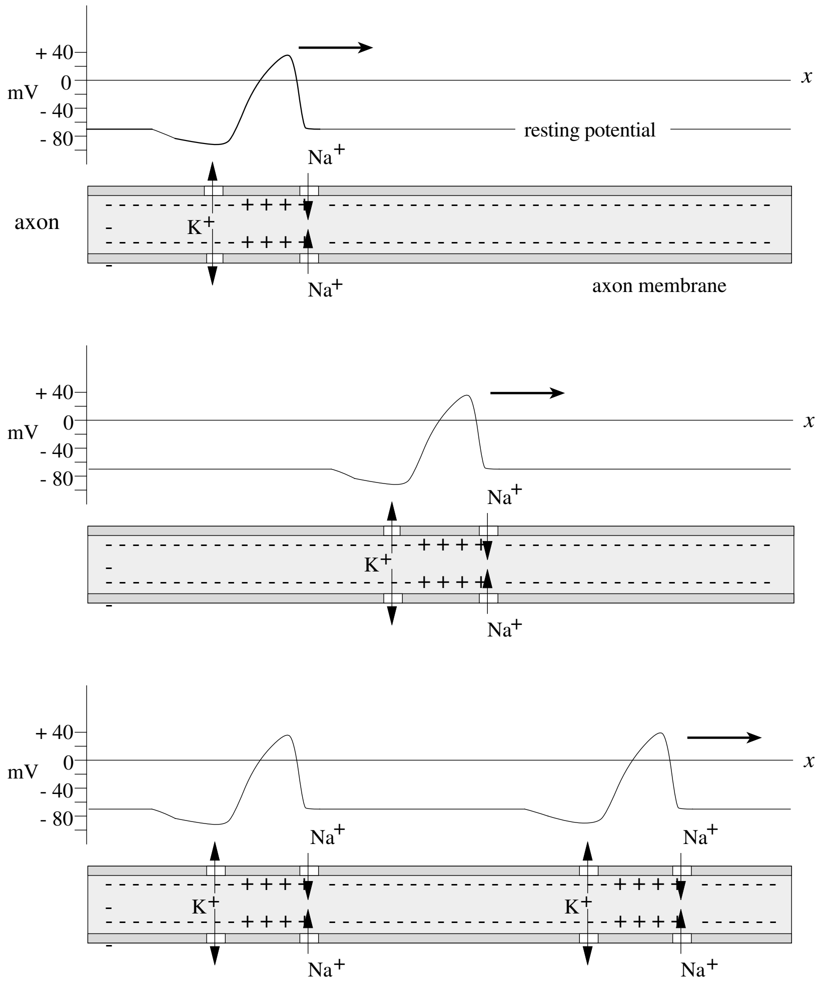


Fig. 1.10. Transmission of an action potential [Stevens 1988]

With this picture of the way an action potential is generated in mind, it is easy to understand the celebrated Hodgkin–Huxley differential equation which describes the instantaneous variation of the cell's potential V as a function of the conductances of sodium, potassium and leakages $(g_{\mathrm{Na}}, g_{\mathrm{K}}, g_{\mathrm{L}})$ and of the equilibrium potentials for all three groups of ions called $V_{\mathrm{Na}}, V_{\mathrm{K}}$ and V_{L} with respect to the current potential:

$$\frac{dV}{dt} = \frac{1}{C_{\mathrm{m}}}(I - g_{\mathrm{Na}}(V - V_{\mathrm{Na}}) - g_{\mathrm{K}}(V - V_{\mathrm{K}}) - g_{\mathrm{L}}(V - V_{\mathrm{L}})). \tag{1.1}$$

In this equation C_{m} is the capacitance of the cell membrane. The terms $V - V_{\mathrm{Na}}$, $V - V_{\mathrm{K}}$, $V - V_{\mathrm{L}}$ are the electromotive forces acting on the ions. Any variation of the conductances translates into a corresponding variation of the cell's potential V. The variations of g_{Na} and g_{K} are given by differential equations which describe their oscillations. The conductance of the leakages, g_{L}, can be taken as a constant.

A neuron codes its level of activity by adjusting the frequency of the generated impulses. This frequency is greater for a greater stimulus. In some cells the mapping from stimulus to frequency is linear in a certain interval [72]. This means that information is transmitted from cell to cell using what engineers call frequency modulation. This form of transmission helps to increase the accuracy of the signal and to minimize the energy consumption of the cells.

1.2.3 Information processing at the neurons and synapses

Neurons transmit information using action potentials. The processing of this information involves a combination of electrical and chemical processes, regulated for the most part at the interface between neurons, the synapses.

Neurons transmit information not only by electrical perturbations. Although electrical synapses are also known, most synapses make use of chemical signaling. Figure 1.11 is a classical diagram of a typical synapse. The synapse appears as a thickening of the axon. The small vacuoles in the interior, the synaptic vesicles, contain chemical transmitters. The small gap between a synapse and the cell to which it is attached is known as the synaptic gap.

When an electric impulse arrives at a synapse, the synaptic vesicles fuse with the cell membrane (Figure 1.12). The transmitters flow into the synaptic gap and some attach themselves to the ionic channels, as in our example. If the transmitter is of the right kind, the ionic channels are opened and more ions can now flow from the exterior to the interior of the cell. The cell's potential is altered in this way. If the potential in the interior of the cell is increased, this helps prepare an action potential and the synapse causes an excitation of the cell. If negative ions are transported into the cell, the probability of starting an action potential is decreased for some time and we are dealing with an inhibitory synapse.

Synapses determine a direction for the transmission of information. Signals flow from one cell to the other in a well-defined manner. This will be expressed in artificial neural networks models by embedding the computing elements in a

Fig. 1.11. Transversal view of a synapse [from Stevens 1988]

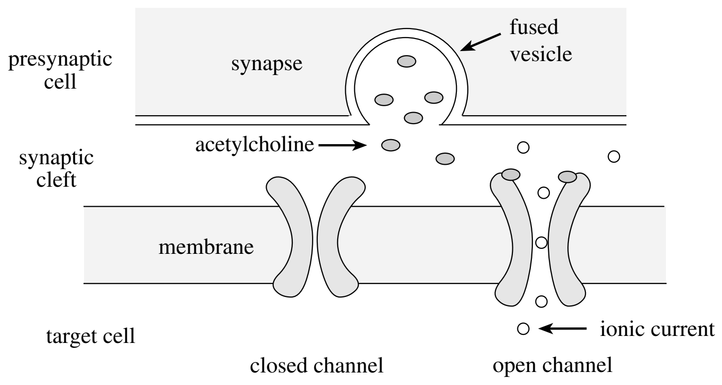


Fig. 1.12. Chemical signaling at the synapse

directed graph. A well-defined direction of information flow is a basic element in every computing model, and is implemented in digital systems by using diodes and directional amplifiers.

The interplay between electrical transmission of information in the cell and chemical transmission between cells is the basis for neural information processing. Cells process information by integrating incoming signals and by reacting to inhibition. The flow of transmitters from an excitatory synapse leads to a depolarization of the attached cell. The depolarization must exceed a threshold, that is, enough ionic channels have to be opened in order to produce an action potential. This can be achieved by several pulses arriving simultaneously or within a short time interval at the cell. If the quantity of transmitters reaches a certain level and enough ionic channels are triggered, the cell reaches its activation threshold. At this moment an action potential is generated at the axon of this cell.

In most neurons, action potentials are produced at the so-called axon hillock, the part of the axon nearest to the cell body. In this region of the cell, the number of ionic channels is larger and the cell's threshold lower [427]. The dendrites collect the electrical signals which are then transmitted electrotonically, that is through the cytoplasm [420]. The transmission of information at the dendrites makes use of additional electrical effects. Streams of ions are collected at the dendrites and brought to the axon hillock. There is spatial summation of information when signals coming from different dendrites are collected, and temporal summation when signals arriving consecutively are combined to produce a single reaction. In some neurons not only the axon hillock but also the dendrites can produce action potentials. In this case information processing at the cell is more complex than in the standard case.

It can be shown that digital signals combined in an excitatory or inhibitory way can be used to implement any desired logical function (Chap. 2). The number of computing units required can be reduced if the information is not only transmitted but also weighted. This can be achieved by multiplying the signal by a constant. Such is the kind of processing we find at the synapses. Each signal is an all-or-none event but the number of ionic channels triggered by the signal is different from synapse to synapse. It can happen that a single synapse can push a cell to fire an action potential, but other synapses can achieve this only by simultaneously exciting the cell. With each synapse i ($1 \leq i \leq n$) we can therefore associate a numerical weight w_i. If all synapses are activated at the same time, the information which will be transmitted is $w_1 + w_2 + \cdots + w_n$. If this value is greater than the cell's threshold, the cell will fire a pulse.

It follows from this description that neurons process information at the membrane. The membrane regulates both transmission and processing of information. Summation of signals and comparison with a threshold is a combined effect of the membrane and the cytoplasm. If a pulse is generated, it is transmitted and the synapses set some transmitter molecules free. From this description an *abstract neuron* [72] can be modeled which contains dendrites, a cell body and an axon. The same three elements will be present in our artificial computing units.

1.2.4 Storage of information – learning

In neural networks information is stored at the synapses. Some other forms of information storage may be present, but they are either still unknown or not very well understood.

A synapse's efficiency in eliciting the depolarization of the contacted cell can be increased if more ionic channels are opened. In recent years NMDA receptors have been studied because they exhibit some properties which could help explain some forms of learning in neurons [72].

NMDA receptors are ionic channels permeable for different kinds of molecules, like sodium, calcium, or potassium ions. These channels are blocked by a magnesium ion in such a way that the permeability for sodium and calcium is low. If the cell is brought up to a certain excitation level, the ionic channels lose the magnesium ion and become unblocked. The permeability for Ca^{2+} ions increases immediately. Through the flow of calcium ions a chain of reactions is started which produces a durable change of the threshold level of the cell [420, 360]. Figure 1.13 shows a diagram of this process.

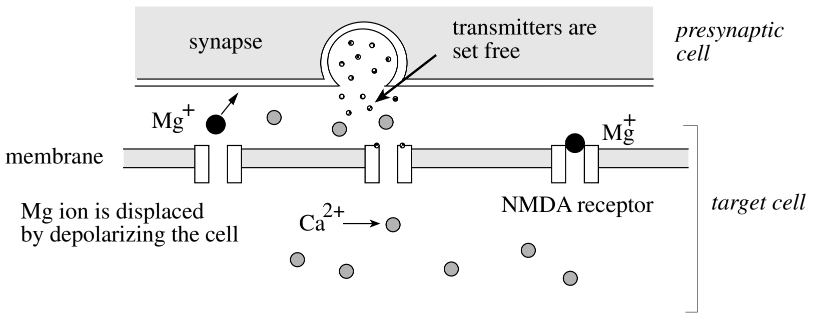

synapse
transmitters are set free
presynaptic cell
Mg+
Mg+
membrane
Mg ion is displaced by depolarizing the cell
Ca2+
NMDA receptor
target cell

Fig. 1.13. Unblocking of an NMDA receptor

NMDA receptors are just one of the mechanisms used by neurons to increase their plasticity, i.e., their adaptability to changing circumstances. Through the modification of the membrane's permeability a cell can be trained to fire more often by setting a lower firing threshold. NMDA receptors also offer an explanation for the observed phenomenon that cells which are not stimulated to fire tend to set a higher firing threshold. The stored information must be refreshed periodically in order to maintain the optimal permeability of the cell membrane.

This kind of information storage is also used in artificial neural networks. Synaptic efficiency can be modeled as a property of the edges of the network. The networks of neurons are thus connected through edges with different transmission efficiencies. Information flowing through the edges is multiplied by a constant which reflects their efficiency. One of the most popular learning algorithms for artificial neural networks is *Hebbian learning*. The efficiency of synapses is increased any time the two cells which are connected through this synapse fire simultaneously and is decreased when the firing states of the two cells are uncorrelated. The NMDA receptors act as coincidence detectors of presynaptic and postsynaptic activity, which in turn leads to greater synaptic efficiency.

1.2.5 The neuron – a self-organizing system

The short review of the properties of biological neurons in the previous sections is necessarily incomplete and can offer only a rough description of the mechanisms and processes by which neurons deal with information. Nerve cells are very complex self-organizing systems which have evolved in the course of millions of years. How were these exquisitely fine-tuned information processing organs developed? Where do we find the evolutionary origin of consciousness?

The information processing capabilities of neurons depend essentially on the characteristics of the cell membrane. Ionic channels appeared very early in evolution to allow unicellular organisms to get some kind of feedback from the environment. Consider the case of a paramecium, a protozoan with cilia, which are hairlike processes which provide it with locomotion. A paramecium has a membrane cell with ionic channels and its normal state is one in which the interior of the cell is negative with respect to the exterior. In this state the cilia around the membrane beat rhythmically and propel the paramecium forward. If an obstacle is encountered, some ionic channels sensitive to contact open, let ions into the cell, and depolarize it. The depolarization of the cell leads in turn to a reversing of the beating direction of the cilia and the paramecium swims backward for a short time. After the cytoplasm returns to its normal state, the paramecium swims forward, changing its direction of movement. If the paramecium is touched from behind, the opening of ionic channels leads to a forward acceleration of the protozoan. In each case, the paramecium escapes its enemies [190].

From these humble origins, ionic channels in neurons have been perfected over millions of years of evolution. In the protoplasm of the cell, ionic channels are produced and replaced continually. They attach themselves to those regions of the neurons where they are needed and can move laterally in the membrane, like icebergs in the sea. The regions of increased neural sensitivity to the production of action potentials are thus changing continuously according to experience. The electrical properties of the cell membrane are not totally predetermined. They are also a result of the process by which action potentials are generated.

Consider also the interior of the neurons. The number of biochemical reaction chains and the complexity of the mechanical processes occurring in the neuron at any given time have led some authors to look for its *control system*. Stuart Hameroff, for example, has proposed that the cytoskeleton of neurons does not just perform a static mechanical function, but in some way provides the cell with feedback control. It is well known that the proteins that form the microtubules in axons coordinate to move synaptic vesicles and other materials from the cell body to the synapses. This is accomplished through a coordinated movement of the proteins, configured like a cellular automaton [173, 174].

Consequently, transmission, storage, and processing of information are performed by neurons exploiting many effects and mechanisms which we still do not understand fully. Each individual neuron is as complex or more complex than any of our computers. For this reason, we will call the elementary components of artificial neural networks simply "computing units" and not neurons. In the mid-1980s, the PDP (*Parallel Distributed Processing*) group already agreed to this convention at the insistence of Francis Crick [95].

1.3 Artificial neural networks

The discussion in the last section is only an example of how important it is to define the primitive functions and composition rules of the computational model. If we are computing with a conventional von Neumann processor, a minimal set of machine instructions is needed in order to implement all computable functions. In the case of artificial neural networks, the primitive functions are located in the nodes of the network and the composition rules are contained implicitly in the interconnection pattern of the nodes, in the synchrony or asynchrony of the transmission of information, and in the presence or absence of cycles.

1.3.1 Networks of primitive functions

Figure 1.14 shows the structure of an abstract neuron with n inputs. Each input channel i can transmit a real value x_i. The *primitive function* f computed in the body of the abstract neuron can be selected arbitrarily. Usually the input channels have an associated weight, which means that the incoming information x_i is multiplied by the corresponding weight w_i. The transmitted information is integrated at the neuron (usually just by adding the different signals) and the primitive function is then evaluated.

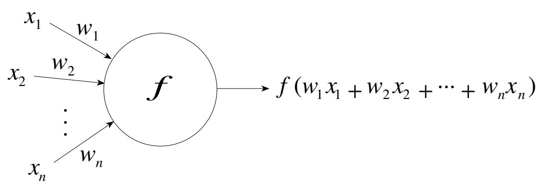


Fig. 1.14. An abstract neuron

If we conceive of each node in an artificial neural network as a primitive function capable of transforming its input in a precisely defined output, then artificial neural networks are nothing but *networks of primitive functions.* Different models of artificial neural networks differ mainly in the assumptions about the primitive functions used, the interconnection pattern, and the timing of the transmission of information.

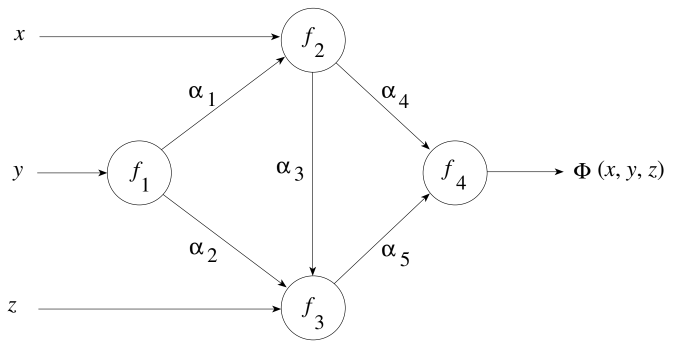


Fig. 1.15. Functional model of an artificial neural network

Typical artificial neural networks have the structure shown in Figure 1.15. The network can be thought of as a function Φ which is evaluated at the point (x, y, z). The nodes implement the primitive functions f_1, f_2, f_3, f_4 which are combined to produce Φ. The function Φ implemented by a neural network will be called the *network function*. Different selections of the weights $\alpha_1, \alpha_2, \ldots, \alpha_5$ produce different network functions. Therefore, tree elements are particularly important in any model of artificial neural networks:

- the structure of the nodes,
- the topology of the network,
- the learning algorithm used to find the weights of the network.

To emphasize our view of neural networks as networks of functions, the next section gives a short preview of some of the topics covered later in the book.

1.3.2 Approximation of functions

An old problem in approximation theory is to reproduce a given function $F : \mathbb{R} \to \mathbb{R}$ either exactly or approximately by evaluating a given set of primitive functions. A classical example is the approximation of one-dimensional functions using polynomials or Fourier series. The Taylor series for a function F which is being approximated around the point x_0 is

$$F(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)^2 + \cdots + a_n(x - x_0)^n + \cdots,$$

whereby the constants $a_0, ..., a_n$ depend on the function F and its derivatives at x_0. Figure 1.16 shows how the polynomial approximation can be represented as a network of functions. The primitive functions $z \mapsto 1, z \mapsto z^1, \ldots, z \mapsto z^n$ are computed at the nodes. The only free parameters are the constants $a_0, ..., a_n$. The output node additively collects all incoming information and produces the value of the evaluated polynomial. The weights of the network can be calculated in this case analytically, just by computing the first $n + 1$

terms of the Taylor series of F. They can also be computed using a learning algorithm, which is the usual case in the field of artificial neural networks.

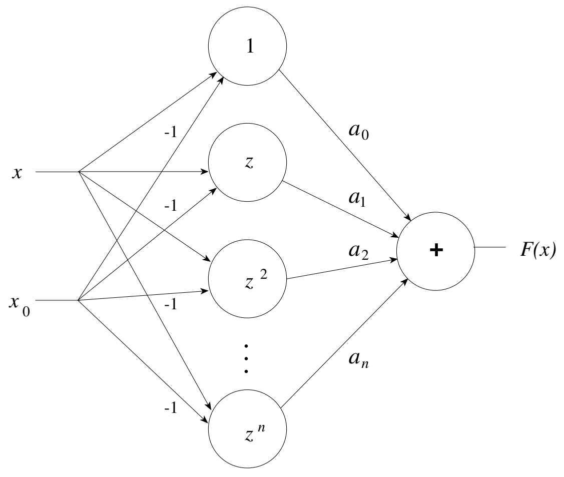


Fig. 1.16. A Taylor network

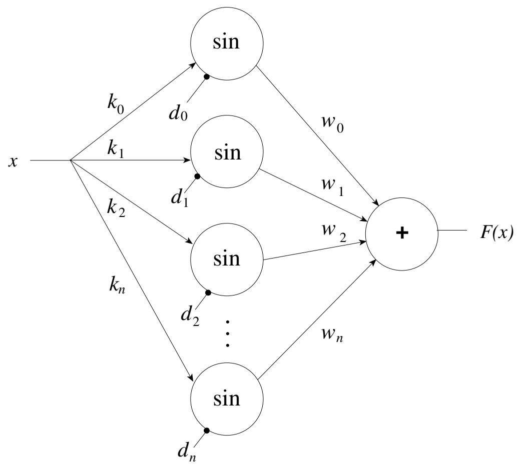


Fig. 1.17. A Fourier network

Figure 1.17 shows how a Fourier series can be implemented as a neural network. If the function F is to be developed as a Fourier series it has the form

$$F(x) = \sum_{i=0}^{\infty} (a_i \cos(ix) + b_i \sin(ix)). \tag{1.2}$$

An artificial neural network with the sine as primitive function can implement a finite number of terms in the above expression. In Figure 1.17 the constants $k_0, \ldots, k_n$ determine the wave numbers for the arguments of the sine functions. The constants $d_0, \ldots, d_n$ play the role of phase factors (with $d_0 = \pi/2$, for example, we have $\sin(x + d_0) = \cos(x)$) and we do not need to implement the cosine explicitly in the network. The constants $w_0, \ldots, w_n$ are the amplitudes of the Fourier terms. The network is indeed more general than the conventional formula because non-integer wave numbers are allowed as are phase factors which are not simple integer multiples of $\pi/2$.

The main difference between Taylor or Fourier series and artificial neural networks is, however, that the function F to be approximated is given not explicitly but implicitly through a set of input-output examples. We know F only at some points but we want to generalize as well as possible. This means that we try to adjust the parameters of the network in an optimal manner to reflect the information known and to extrapolate to new input patterns which will be shown to the network afterwards. This is the task of the *learning algorithm* used to adjust the network's parameters.

These two simple examples show that neural networks can be used as universal function approximators, that is, as computing models capable of approximating a given set of functions (usually the integrable functions). We will come back to this problem in Chap. 10.

1.3.3 Caveat

At this point we must issue a warning to the reader: in the theory of artificial neural networks we do not consider the whole complexity of real biological neurons. We only abstract some general principles and content ourselves with different levels of detail when simulating neural ensembles. The general approach is to conceive each neuron as a primitive function producing numerical results at some points in time. These will be the kinds of model that we will discuss in the first chapters of this book. However we can also think of artificial neurons as computing units which produce pulse trains in the way that biological neurons do. We can then simulate this behavior and look at the output of simple networks. This kind of approach, although more closely related to the biological paradigm, is still a very rough approximation of the biological processes. We will deal with asynchronous and spiking neurons in later chapters.

1.4 Historical and bibliographical remarks

Philosophical reflection on consciousness and the organ in which it could possibly be localized spans a period of more than two thousand years. Greek philosophers were among the first to speculate about the location of the soul. Several theories were held by the various philosophical schools of ancient times. Galenus, for example, identified nerve impulses with pneumatic pressure signals and conceived the nervous system as a pneumatic machine. Several centuries later Newton speculated that nerves transmitted oscillations of the ether.

Our present knowledge of the structure and physiology of neurons is the result of 100 years of special research in this field. The facts presented in this chapter were discovered between 1850 and 1950, with the exception of the NMDA receptors which were studied mainly in the last decade. The electrical nature of nerve impulses was postulated around 1850 by Emil du Bois-Reymond and Hermann von Helmholtz. The latter was able to measure the velocity of nerve impulses and showed that it was not as fast as was previously thought. Signals can be transmitted in both directions of an axon, but around 1901 Santiago Ramón y Cajal postulated that the specific networking of the nervous cells determines a direction for the transmission of information. This discovery made it clear that the coupling of the neurons constitutes a hierarchical system.

Ramón y Cajal was also the most celebrated advocate of the *neuron theory.* His supporters conceived the brain as a highly differentiated hierarchical organ, while the supporters of the reticular theory thought of the brain as a grid of undifferentiated axons and of dendrites as organs for the nutrition of the cell [357]. Ramón y Cajal perfected Golgi's staining method and published the best diagrams of neurons of his time, so good indeed that they are still in use. The word *neuron* (Greek for *nerve*) was proposed by the Berlin Professor Wilhelm Waldeger after he saw the preparations of Ramón y Cajal [418].

The chemical transmission of information at the synapses was studied from 1920 to 1940. From 1949 to 1956, Hodgkin and Huxley explained the mechanism by which depolarization waves are produced in the cell membrane. By experimenting with the giant axon of the squid they measured and explained the exchange of ions through the cell membrane, which in time led to the now famous Hodgkin–Huxley differential equations. For a mathematical treatment of this system of equations see [97].

The Hodgkin–Huxley model was in some ways one of the first artificial neural models, because the postulated dynamics of the nerve impulses could be simulated with simple electric networks [303]. At the same time the mathematical properties of artificial neural networks were being studied by researchers like Warren McCulloch, Walter Pitts, and John von Neumann. Ever since that time, research in the neurobiological field has progressed in close collaboration with the mathematics and computer science community.

Exercises

1. Express the network function function Φ in Figure 1.15 in terms of the primitive functions $f_1, \ldots, f_4$ and of the weights $\alpha_1, \ldots, \alpha_5$.

2. Modify the network of Figure 1.17 so that it corresponds to a finite number of addition terms of equation (1.2).
3. Look in a neurobiology book for the full set of differential equations of the Hodgkin–Huxley model. Write a computer program that simulates an action potential.

2

Threshold Logic

2.1 Networks of functions

We deal in this chapter with the simplest kind of computing units used to build artificial neural networks. These computing elements are a generalization of the common logic gates used in conventional computing and, since they operate by comparing their total input with a threshold, this field of research is known as *threshold logic*.

2.1.1 Feed-forward and recurrent networks

Our review in the previous chapter of the characteristics and structure of biological neural networks provides us with the initial motivation for a deeper inquiry into the properties of networks of abstract neurons. From the viewpoint of the engineer, it is important to define how a network should behave, without having to specify completely all of its parameters, which are to be found in a learning process. Artificial neural networks are used in many cases as a *black box*: a certain input should produce a desired output, but how the network achieves this result is left to a self-organizing process.

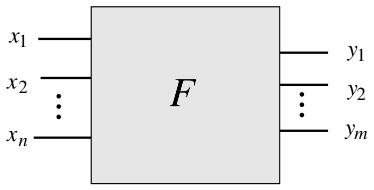


Fig. 2.1. A neural network as a black box

In general we are interested in mapping an n-dimensional real input $(x_1, x_2, \ldots, x_n)$ to an m-dimensional real output $(y_1, y_2, \ldots, y_m)$. A neural network thus behaves as a “mapping machine”, capable of modeling a function $F : \mathbb{R}^n \to \mathbb{R}^m$. If we look at the structure of the network being used, some aspects of its dynamics must be defined more precisely. When the function is evaluated with a network of primitive functions, information flows through the directed edges of the network. Some nodes compute values which are then transmitted as arguments for new computations. If there are no cycles in the network, the result of the whole computation is well-defined and we do not have to deal with the task of synchronizing the computing units. We just assume that the computations take place without delay.

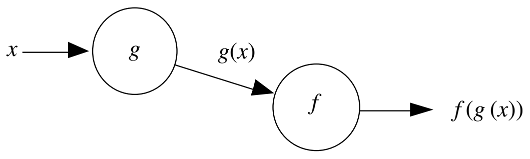


Fig. 2.2. Function composition

If the network contains cycles, however, the computation is not uniquely defined by the interconnection pattern and the temporal dimension must be considered. When the output of a unit is fed back to the same unit, we are dealing with a recursive computation without an explicit halting condition. We must define what we expect from the network: is the fixed point of the recursive evaluation the desired result or one of the intermediate computations? To solve this problem we assume that every computation takes a certain amount of time at each node (for example a time unit). If the arguments for a unit have been transmitted at time t, its output will be produced at time $t + 1$. A recursive computation can be stopped after a certain number of steps and the last computed output taken as the result of the recursive computation.

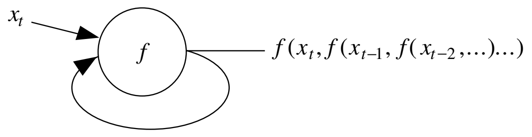


Fig. 2.3. Recursive evaluation

In this chapter we deal first with networks without cycles, in which the time dimension can be disregarded. Then we deal with recurrent networks and their temporal coordination. The first model we consider was proposed in 1943 by Warren McCulloch and Walter Pitts. Inspired by neurobiology they put forward a model of computation oriented towards the computational capabilities of real neurons and studied the question of abstracting universal concepts from specific perceptions [299].

We will avoid giving a general definition of a *neural network* at this point. So many models have been proposed which differ in so many respects that any definition trying to encompass this variety would be unnecessarily clumsy. As we show in this chapter, it is not necessary to start building neural networks with "high powered" computing units, as some authors do [384]. We will start our investigations with the general notion that a neural network is a *network of functions* in which synchronization can be considered explicitly or not.

2.1.2 The computing units

The nodes of the networks we consider will be called *computing elements* or simply *units*. We assume that the edges of the network transmit information in a predetermined direction and the number of incoming edges into a node is not restricted by some upper bound. This is called the *unlimited fan-in* property of our computing units.

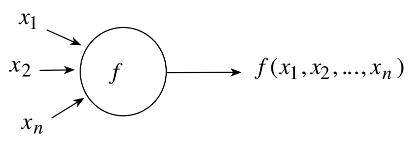


Fig. 2.4. Evaluation of a function of n arguments

The primitive function computed at each node is in general a function of n arguments. Normally, however, we try to use very simple primitive functions of one argument at the nodes. This means that the incoming n arguments have to be reduced to a single numerical value. Therefore computing units are split into two functional parts: an integration function g reduces the n arguments to a single value and the output or activation function f produces the output of this node taking that single value as its argument. Figure 2.5 shows this general structure of the computing units. Usually the integration function g is the addition function.

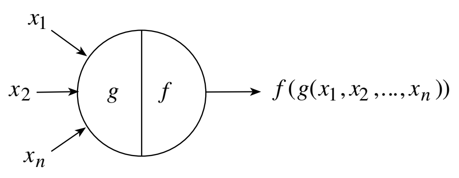


Fig. 2.5. Generic computing unit

McCulloch–Pitts networks are even simpler than this, because they use solely binary signals, i.e., ones or zeros. The nodes produce only binary results and the edges transmit exclusively ones or zeros. The networks are composed of directed unweighted edges of *excitatory* or of *inhibitory* type. The latter are marked in diagrams using a small circle attached to the end of the edge. Each McCulloch–Pitts unit is also provided with a certain threshold value θ.

At first sight the McCulloch–Pitts model seems very limited, since only binary information can be produced and transmitted, but it already contains all necessary features to implement the more complex models. Figure 2.6 shows an abstract McCulloch–Pitts computing unit. Following Minsky [311] it will be represented as a circle with a black half. Incoming edges arrive at the white half, outgoing edges leave from the black half. Outgoing edges can fan out any number of times.

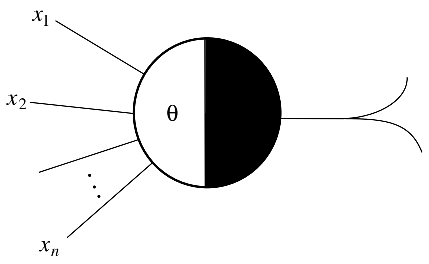


Fig. 2.6. Diagram of a McCulloch–Pitts unit

The rule for evaluating the input to a McCulloch–Pitts unit is the following:

- Assume that a McCulloch–Pitts unit gets an input $x_1, x_2, \ldots, x_n$ through n excitatory edges and an input $y_1, y_2, \ldots, y_m$ through m inhibitory edges.
- If $m \geq 1$ and at least one of the signals $y_1, y_2, \ldots, y_m$ is 1, the unit is inhibited and the result of the computation is 0.
- Otherwise the total excitation $x = x_1 + x_2 + \cdots + x_n$ is computed and compared with the threshold θ of the unit (if $n = 0$ then $x = 0$). If $x \geq \theta$ the unit *fires* a 1, if $x < \theta$ the result of the computation is 0.

This rule implies that a McCulloch–Pitts unit can be inactivated by a single inhibitory signal, as is the case with some real neurons. When no inhibitory signals are present, the units act as a *threshold gate* capable of implementing many other logical functions of n arguments.

Figure 2.7 shows the activation function of a unit, the so-called step function. This function changes discontinuously from zero to one at θ. When θ is zero and no inhibitory signals are present, we have the case of a unit producing the constant output one. If θ is greater than the number of incoming excitatory edges, the unit will never fire.

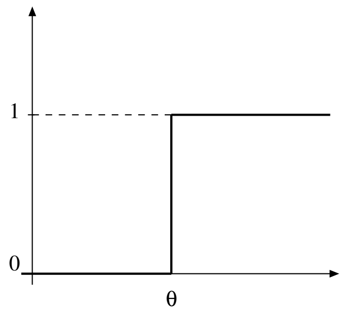


Fig. 2.7. The step function with threshold θ

In the following subsection we assume provisionally that there is no delay in the computation of the output.

2.2 Synthesis of Boolean functions

The power of threshold gates of the McCulloch–Pitts type can be illustrated by showing how to synthesize any given logical function of n arguments. We deal firstly with the more simple kind of logic gates.

2.2.1 Conjunction, disjunction, negation

Mappings from $\{0,1\}^n$ onto $\{0,1\}$ are called logical or Boolean functions. Simple logical functions can be implemented directly with a single McCulloch–Pitts unit. The output value 1 can be associated with the logical value *true* and 0 with the logical value *false*. It is straightforward to verify that the two units of Figure 2.8 compute the functions AND and OR respectively.

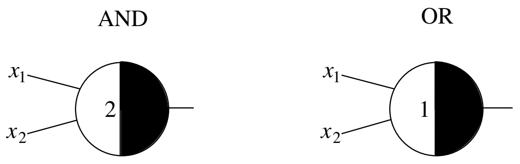


Fig. 2.8. Implementation of AND and OR gates

A single unit can compute the disjunction or the conjunction of n arguments as is shown in Figure 2.9, where the conjunction of three and four arguments is computed by two units. The same kind of computation requires several conventional logic gates with two inputs. It should be clear from this simple example that threshold logic elements can reduce the complexity of the circuit used to implement a given logical function.

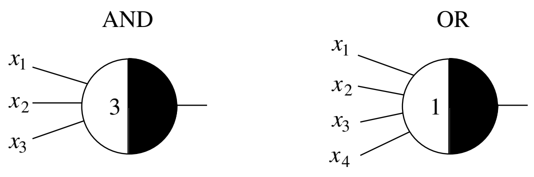


Fig. 2.9. Generalized AND and OR gates

As is well known, AND and OR gates alone cannot be combined to produce all logical functions of n variables. Since uninhibited threshold logic elements are capable of implementing more general functions than conventional AND or OR gates, the question of whether they can be combined to produce all logical functions arises. Stated another way: is inhibition of McCulloch–Pitts units necessary or can it be dispensed with? The following proposition shows that it is necessary. A monotonic logical function f of n arguments is one whose value at two given n-dimensional points $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$ is such that $f(x) \geq f(y)$ whenever the number of ones in the input y is a subset of the ones in the input x. An example of a non-monotonic logical function of one argument is logical negation.

Proposition 1. *Uninhibited threshold logic elements of the McCulloch–Pitts type can only implement monotonic logical functions.*

Proof. An example shows the kind of argumentation needed. Assume that the input vector $(1, 1, \ldots, 1)$ is assigned the function value 0. Since no other vector can set more edges in the network to 1 than this vector does, any other input vector can also only be evaluated to 0. In general, if the ones in the input vector y are a subset of the ones in the input vector x, then the first cannot set more edges to 1 than x does. This implies that $f(x) \geq f(y)$, as had to be shown. □

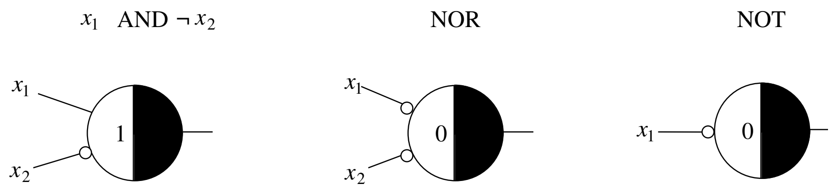


Fig. 2.10. Logical functions and their realization

The units of Figure 2.10 show the implementation of some non-monotonic logical functions requiring inhibitory connections. Logical negation, for example, can be computed using a McCulloch–Pitts unit with threshold 0 and an inhibitory edge. The other two functions can be verified by the reader.

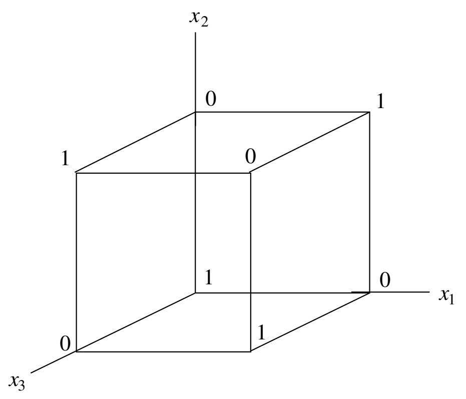


Fig. 2.11. Function values of a logical function of three variables

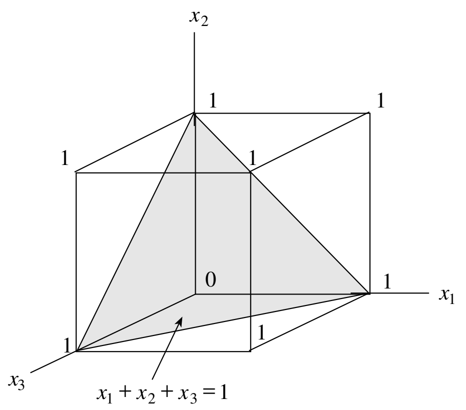


Fig. 2.12. Separation of the input space for the OR function

2.2.2 Geometric interpretation

It is very instructive to visualize the kind of functions that can be computed with McCulloch–Pitts cells by using a diagram. Figure 2.11 shows the eight vertices of a three-dimensional unit cube. Each of the three logical variables x_1, x_2 and x_3 can assume one of two possible binary values. There are eight possible combinations, represented by the vertices of the cube. A logical function is just an assignment of a 0 or a 1 to each of the vertices. The figure shows one of these assignments. In the case of n variables, the cube consists of 2^n vertices and admits 2^{2^n} different binary assignments.

McCulloch–Pitts units divide the input space into two half-spaces. For a given input (x_1, x_2, x_3) and a threshold θ the condition $x_1 + x_2 + x_3 \geq \theta$ is tested, which is true for all points to one side of the plane with the equation $x_1 + x_2 + x_3 = \theta$ and false for all points to the other side (without including the plane itself in this case). Figure 2.12 shows this separation for the case in

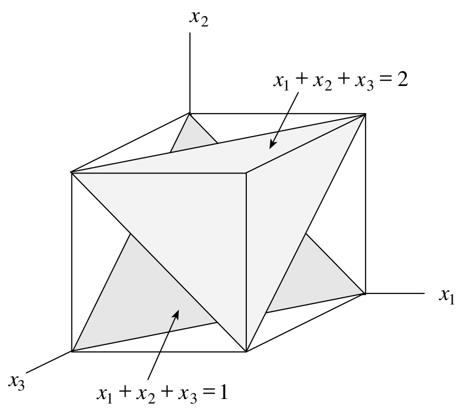


Fig. 2.13. Separating planes of the OR and majority functions

which $\theta = 1$, i.e., for the OR function. Only those vertices above the separating plane are labeled 1.

The majority function of three variables divides input space in a similar manner, but the separating plane is given by the equation $x_1 + x_2 + x_3 = 2$. Figure 2.13 shows the additional plane. The planes are always parallel in the case of McCulloch–Pitts units. Non-parallel separating planes can only be produced using weighted edges.

2.2.3 Constructive synthesis

Every logical function of n variables can be written in tabular form. The value of the function is written down for every one of the possible binary combinations of the n inputs. If we want to build a network to compute this function, it should have n inputs and one output. The network must associate each input vector with the correct output value. If the number of computing units is not limited in some way, it is always possible to build or synthesize a network which computes this function. The constructive proof of this proposition profits from the fact that McCulloch–Pitts units can be used as binary decoders.

Consider for example the vector $(1, 0, 1)$. It is the only one which fulfills the condition $x_1 \wedge \neg x_2 \wedge x_3$. This condition can be tested by a single computing unit (Figure 2.14). Since only the vector $(1, 0, 1)$ makes this unit fire, the unit is a decoder for this input.

Assume that a function F of three arguments has been defined according to the following table:

To compute this function it is only necessary to decode all those vectors for which the function's value is 1. Figure 2.15 shows a network capable of computing the function F.

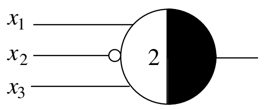


Fig. 2.14. Decoder for the vector $(1, 0, 1)$

input vectors	F
(0,0,1)	1
(0,1,0)	1
all others	0

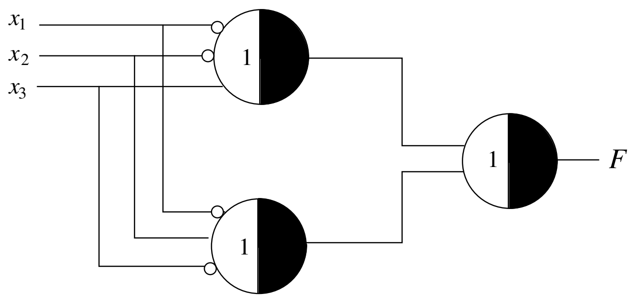


Fig. 2.15. Synthesis of the function F

The individual units in the first layer of the composite network are decoders. For each vector for which F is 1 a decoder is used. In our case we need just two decoders. Components of each vector which must be 0 are transmitted with inhibitory edges, components which must be 1 with excitatory ones. The threshold of each unit is equal to the number of bits equal to 1 that must be present in the desired input vector. The last unit to the right is a disjunction: if any one of the specified vectors can be decoded this unit fires a 1.

It is straightforward to extend this constructive method to other Boolean functions of any other dimension. This leads to the following proposition:

Proposition 2. *Any logical function* $F : \{0, 1\}^n \rightarrow \{0, 1\}$ *can be computed with a McCulloch–Pitts network of two layers.*

No attempt has been made here to minimize the number of computing units. In fact, we need as many decoders as there are ones in the table of function values. An alternative to this simple constructive method is to use harmonic analysis of logical functions, as will be shown in Sect. 2.5.

We can also consider the minimal possible set of building blocks needed to implement arbitrary logical functions when the fan-in of the units is bounded in some way. The circuits of Figure 2.14 and Figure 2.15 use decoders of n inputs. These decoders can be built of simpler cells, for example, two units capable of respectively implementing the AND function and negation. Inhibitory connections in the decoders can be replaced with a negation gate. The output of the decoders is collected at a conjunctive unit. The decoder of Figure 2.14 can be implemented as shown in Figure 2.16. The only difference from the previous decoder are the negated inputs and the higher threshold in the AND unit. All decoders for a row of the table of a logical function can be designed in a similar way. This immediately leads to the following proposition:

Proposition 3. *All logical functions can be implemented with a network composed of units which exclusively compute the AND, OR, and NOT functions.*

The three units AND, NOT and OR are called a logical basis because of this property. Since OR can be implemented using AND and NOT units, these two alone constitute a logical basis. The same happens with OR and NOT units. John von Neumann showed that through a redundant coding of the inputs (each variable is transmitted through two lines) AND and OR units alone can constitute a logical basis [326].

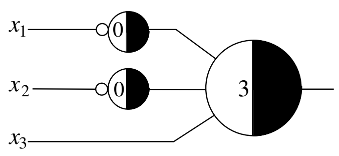


Fig. 2.16. A composite decoder for the vector $(0, 0, 1)$

2.3 Equivalent networks

We can build simpler circuits by using units with more general properties, for example weighted edges and relative inhibition. However, as we show in this section, circuits of McCulloch–Pitts units can emulate circuits built out of high-powered units by exploiting the trade-off between the complexity of the network versus the complexity of the computing units.

2.3.1 Weighted and unweighted networks

Since McCulloch–Pitts networks do not use weighted edges the question of whether weighted networks are more general than unweighted ones must be answered. A simple example shows that both kinds of networks are equivalent.

Assume that three weighted edges converge on the unit shown in Figure 2.17. The unit computes

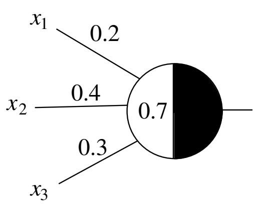


Fig. 2.17. Weighted unit

$$0.2x_1 + 0.4x_2 + 0.3x_3 \geq 0.7.$$

But this is equivalent to

$$2x_1 + 4x_2 + 3x_3 \geq 7,$$

and this computation can be performed with the network of Figure 2.18.

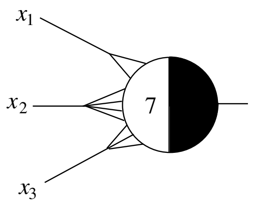


Fig. 2.18. Equivalent computing unit

The figure shows that positive rational weights can be simulated by simply fanning-out the edges of the network the required number of times. This means that we can either use weighted edges or go for a more complex topology of the network, with many redundant edges. The same can be done in the case of irrational weights if the number of input vectors is finite (see Chap. 3, Exercise 3).

2.3.2 Absolute and relative inhibition

In the last subsection we dealt only with the case of positive weights. Two classes of inhibition can be identified: *absolute* inhibition corresponds to the one used in McCulloch–Pitts units. *Relative* inhibition corresponds to the case of edges weighted with a negative factor and whose effect is to lower the firing threshold when a 1 is transmitted through this edge.

Proposition 4. *Networks of McCulloch–Pitts units are equivalent to networks with relative inhibition.*

Proof. It is only necessary to show that each unit in a network where relative inhibition is used is equivalent to one or more units in a network where absolute inhibition is used. It is clear that it is possible to implement absolute inhibition with relative inhibitory edges. If the threshold of a unit is the integer m and if n excitatory edges impinge on it, the maximum possible total excitation for this unit is $n - m$. If $m \geq n$ the unit never fires and the inhibitory edge is irrelevant. It suffices to fan out the inhibitory edge $n - m + 1$ times and make all these edges meet at the unit. When a 1 is transmitted through the inhibitory edges the total amount of inhibition is $n - m + 1$ and this shuts down the unit. To prove that relative inhibitory edges can be simulated with absolute inhibitory ones, refer to Figure 2.19. The network to the left contains a relative inhibitory edge, the network to the right absolute inhibitory ones. The reader can verify that the two networks are equivalent. Relative inhibitory edges correspond to edges weighted with -1. We can also accept any other negative weight w. In that case the threshold of the unit to the right of Figure 2.19 should be $m + w$ instead of $m + 1$. Therefore networks with negative weights can be simulated using unweighted McCulloch–Pitts elements. □

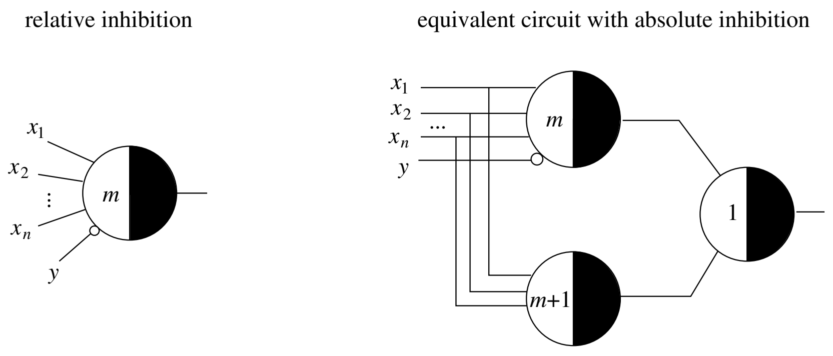


Fig. 2.19. Two equivalent networks

As shown above, we can implement any kind of logical function using unweighted networks. What we trade is the simplicity of the building blocks for a more convoluted topology of the network. Later we will always use weighted networks in order to simplify the topology.

2.3.3 Binary signals and pulse coding

An additional question which can be raised is whether binary signals are not a very limited coding strategy. Are networks in which the communication channels adopt any of ten or fifteen different states more efficient than channels which adopt only two states, as in McCulloch–Pitts networks? To give an answer we must consider that unit states have a price, in biological networks as well as in artificial ones. The transmitted information must be optimized using the number of available switching states.

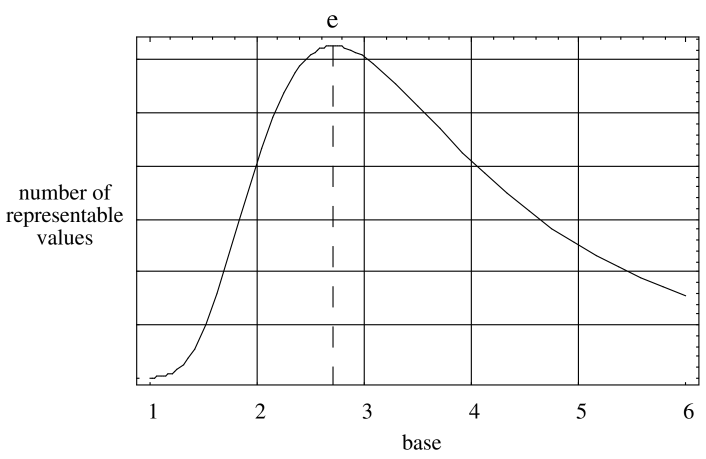


Fig. 2.20. Number of representable values as a function of the base

Assume that the number of states per communication channel is b and that c channels are used to input information. The cost K of the implementation is proportional to both quantities, i.e., $K = \gamma bc$, where γ is a proportionality constant. Using c channels with b states, b^c different numbers can be represented. This means that $c = K/\gamma b$ and, if we set $\kappa = K/\gamma$, we are seeking the numerical base b which optimizes the function $b^{\kappa/b}$. Since we assume constant cost, κ is a constant. Figure 2.20 shows that the optimal value for b is the Euler constant e. Since the number of channel states must be an integer, three states would provide a good approximation to the optimal coding strategy. However, in electronic and biological systems decoding of the signal plays such an important role that the choice of two states per channel becomes a better alternative.

Wiener arrived at a similar conclusion through a somewhat different argument [452]. The binary nature of information transmission in the nervous system seems to be an efficient way to transport signals. However, in the next chapters we will assume that the communication channels can transport *arbitrary real numbers*. This makes the analysis simpler than when we have to deal explicitly with frequency modulated signals, but does not lead to a minimization of the resources needed for a technical implementation. Some researchers prefer to work with so-called *weightless networks* which operate exclusively with binary data.

2.4 Recurrent networks

We have already shown that feed-forward networks can implement arbitrary logical functions. In this case the dimension of the input and output data is predetermined. In many cases, though, we want to perform computations on an input of variable length, for example, when adding two binary numbers being fed bit for bit into a network, which in turn produces the bits of the result one after the other. A feed-forward network cannot solve this problem because it is not capable of keeping track of previous results and, in the case of addition, the carry bit must be stored in order to be reused. This kind of problem can be solved using recurrent networks, i.e., networks whose partial computations are recycled through the network itself. Cycles in the topology of the network make storage and reuse of signals possible for a certain amount of time after they are produced.

2.4.1 Stored state networks

McCulloch–Pitts units can be used in recurrent networks by introducing a temporal factor in the computation. We will assume that computation of the activation of each unit consumes a time unit. If the input arrives at time t the result is produced at time $t + 1$. Up to now, we have been working with units which produce results without delay. The numerical capabilities of any feed-forward network with instantaneous computation at the nodes can be reproduced by networks of units with delay. We only have to take care to coordinate the arrival of the input values at the nodes. This could make the introduction of additional computing elements necessary, whose sole mission is to insert the necessary delays for the coordinated arrival of information. This is the same problem that any computer with clocked elements has to deal with.

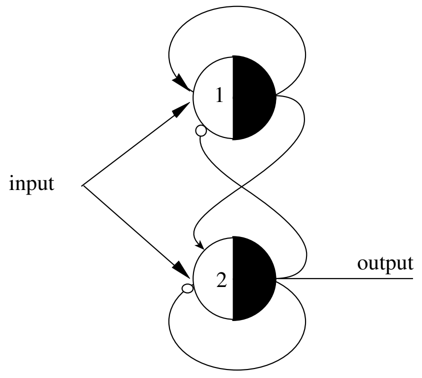


Fig. 2.21. Network for a binary scaler

Figure 2.21 shows a simple example of a recurrent circuit. The network processes a sequence of bits, giving off one bit of output for every bit of input, but in such a way that any two consecutive ones are transformed into the sequence 10. The binary sequence 00110110 is transformed for example into the sequence 00100100. The network recognizes only two consecutive ones separated by at least a zero from a similar block.

2.4.2 Finite automata

The network discussed in the previous subsection is an example of an automaton. This is an abstract device capable of assuming different states which change according to the received input. The automaton also produces an output according to its momentary state. In the previous example, the state of the automaton is the specific combination of signals circulating in the network at any given time. The set of possible states corresponds to the set of all possible combinations of signals traveling through the network.

state transitions

input \ state	Q_0	Q_1
0	Q_0	Q_0
1	Q_1	Q_1

output table

input \ state	Q_0	Q_1
0	0	1
1	0	1

Fig. 2.22. State tables for a binary delay

Finite automata can take only a finite set of possible states and can react only to a finite set of input signals. The state transitions and the output of an automaton can be specified with a table, like the one shown in Figure 2.22. This table defines an automaton which accepts a binary signal at time t and produces an output at time $t+1$. The automaton has two states, Q_0 and Q_1, and accepts only the values 0 or 1. The first table shows the state transitions, corresponding to each input and each state. The second table shows the output values corresponding to the given state and input. From the table we can see that the automaton switches from state Q_0 to state Q_1 after accepting the input 1. If the input bit is a 0, the automaton remains in state Q_0. If the state of the automaton is Q_1 the output at time $t+1$ is 1 regardless of whether 1 or 0 was given as input at time t. All other possibilities are covered by the rest of the entries in the two tables.

The diagram in Figure 2.23 shows how the automaton works. The values at the beginning of the arrows represent an input bit for the automaton. The values in the middle of the arrows are the output bits produced after each new input. An input of 1, for example, produces the transition from state Q_0 to state Q_1 and the output 0. The input 0 produces a transition to state Q_0. The automaton is thus one that stores only the last bit of input in its current state.

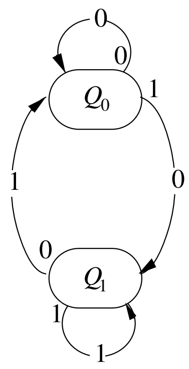


Fig. 2.23. Diagram of a finite automaton

Finite automata without input from the outside, i.e., free-wheeling automata, unavoidably fall in an infinite loop or reach a final constant state. This is why finite automata cannot cover all computable functions, for whose computation an infinite number of states are needed. A Turing machine achieves this through an infinite storage band which provides enough space for all computations. Even a simple problem like the multiplication of two arbitrary binary numbers presented sequentially cannot be solved by a finite automaton. Although our computers are finite automata, the number of possible states is so large that we consider them as universal computing devices for all practical purposes.

2.4.3 Finite automata and recurrent networks

We now show that finite automata and recurrent networks of McCulloch–Pitts units are equivalent. We use a variation of a constructive proof due to Minsky [311].

Proposition 5. *Any finite automaton can be simulated with a network of McCulloch–Pitts units.*

Proof. Figure 2.24 is a diagram of the network needed for the proof. Assume that the input signals are transmitted through the input lines I_1 to I_m and at each moment t only one of these lines is conducting a 1. All other input lines are passive (set to 0). Assume that the network starts in a well-defined

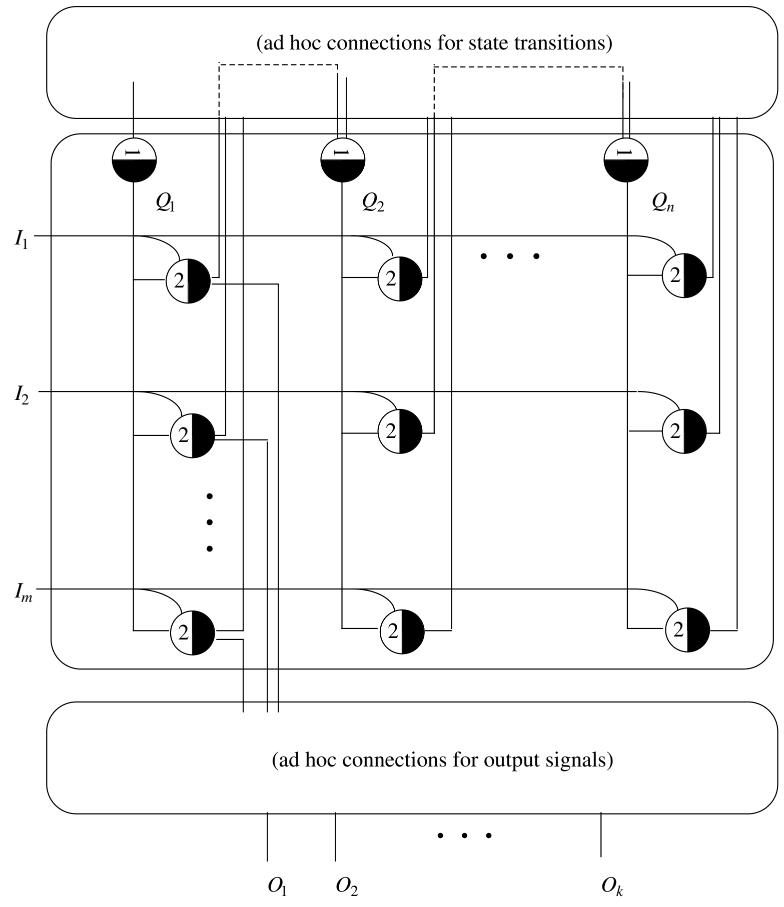

(ad hoc connections for state transitions)
1
Q_1
Q_2
Q_n
I_1
I_2
I_m
2
(ad hoc connections for output signals)
O_1
O_2
O_k

Fig. 2.24. Implementation of a finite automaton with McCulloch–Pitts units

state Q_i. This means that one, and only one, of the lines labeled $Q_1, \ldots, Q_n$ is set to 1 and the others to 0. At time $t+1$ only one of the AND units can produce a 1, namely the one in which both input and state line are set to 1. The state transition is controlled by the ad hoc connections defined by the user in the upper box. If, for example, the input I_1 and the state Q_1 at time t produce the transition to state Q_2 at time $t+1$, then we have to connect the output of the upper left AND unit to the input of the OR unit with the output line named Q_2 (dotted line in the diagram). This output will become active at time $t+2$. At this stage a new input line must be set to 1 (for example I_2) and a new state transition will be computed (Q_n in our example). The connections required to produce the desired output are defined in a similar way. This can be controlled by connecting the output of each AND unit to the corresponding output line $O_1, \ldots, O_k$ using a box of ad hoc connections similar to the one already described. □

A disadvantage of this constructive method is that each simulated finite automaton requires a special set of connections in the upper and lower boxes. It is better to define a universal network capable of simulating any other finite automaton without having to change the topology of the network (under the assumption of an upper bound for the size of the simulated automata). This is indeed an active field of research in which networks learn to simulate automata [408]. The necessary network parameters are found by a learning algorithm. In the case of McCulloch–Pitts units the available degrees of freedom are given by the topology of the network.

2.4.4 A first classification of neural networks

The networks described in this chapter allow us to propose a preliminary taxonomy of the networks we will discuss in this book. The first clear separation line runs between weighted and unweighted networks. It has already been shown that both classes of models are equivalent. The main difference is the kind of learning algorithm that can be used. In unweighted networks only the thresholds and the connectivity can be adapted. In weighted networks the topology is not usually modified during learning (although we will see some algorithms capable of doing this) and only an optimal combination of weights is sought.

The second clear separation is between synchronous and asynchronous models. In synchronous models the output of all elements is computed instantaneously. This is always possible if the topology of the network does not contain cycles. In some cases the models contain layers of computing units and the activity of the units in each layer is computed one after the other, but in each layer simultaneously. Asynchronous models compute the activity of each unit independently of all others and at different stochastically selected times (as in Hopfield networks). In these kinds of models, cycles in the underlying connection graph pose no particular problem.

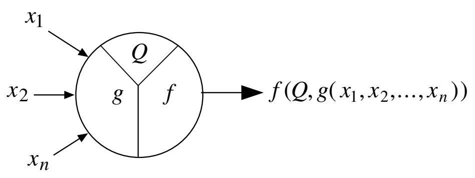


Fig. 2.25. A unit with stored state Q

Finally, we can distinguish between models with or without stored unit states. In Figure 2.5 we gave an example of a unit without stored state. Figure 2.25 shows a unit in which a state Q is stored after each computation.

The state Q can modify the output of the unit in the following activation. If the number of states and possible inputs is finite, we are dealing with a finite automaton. Since any finite automaton can be simulated by a network of computing elements without memory, these units with a stored state can be substituted by a network of McCulloch–Pitts units. Networks with stored-state units are thus equivalent to networks *without* stored-state units. Data is stored in the network itself and in its pattern of recursion.

It can be also shown that time varying weights and thresholds can be implemented in a network of McCulloch–Pitts units using cycles, so that networks with time varying weights and thresholds are equivalent to networks with constant parameters, whenever recursion is allowed.

2.5 Harmonic analysis of logical functions

An interesting alternative for the automatic synthesis of logic functions and for a quantification of their implementation complexity is to do an analysis of the distribution of its non-zero values using the tools of harmonic analysis. Since we can tabulate the values of a logical function in a sequence, we can think of it as a one-dimensional function whose fundamental "frequencies" can be extracted using the appropriate mathematical tools. We will first deal with this problem in a totally general setting and then show that the Hadamard–Walsh transform is the tool we are looking for.

2.5.1 General expression

Assume that we are interested in expressing a function $f : \mathbb{R}^m \to \mathbb{R}$ as a linear combination of n functions $f_1, f_2, \ldots, f_n$ using the n constants $a_1, a_2, \ldots, a_n$ in the following form

$$f = a_1 f_1 + a_2 f_2 + \cdots + a_n f_n.$$

The domain and range of definition are the same for f and the base functions.

We can determine the quadratic error E of the approximation in the whole domain of definition V for given constants $a_1, \ldots, a_n$ by computing

$$E = \int_V (f - (a_1 f_1 + a_2 f_2 + \cdots + a_n f_n))^2 dV.$$

Here we are assuming that f and the functions f_i, $i = 1, \ldots, n$, are integrable in its domain of definition V. Since we want to minimize the quadratic error E we compute the partial derivatives of E with respect to $a_1, a_2, \ldots, a_n$ and set them to zero:

$$\frac{dE}{da_i} = -2 \int_V f_i(f - a_1 f_1 - a_2 f_2 - \cdots - a_n f_n) dV = 0, \text{ for } i = 1, \ldots, n$$

This leads to the following set of n equations expressed in a simplified notation:

$$a_1 \int f_i f_1 + a_2 \int f_i f_2 + \cdots + a_n \int f_i f_n = \int f_i f, \text{ for } i = 1, \ldots, n.$$

Expressed in matrix form the set of equations becomes:

$$\begin{pmatrix} \int f_1 f_1 & \int f_1 f_2 & \cdots & \int f_1 f_n \\ \int f_2 f_1 & \int f_2 f_2 & & \int f_2 f_n \\ \vdots & & \ddots & \vdots \\ \int f_n f_1 & \int f_n f_2 & \cdots & \int f_n f_n \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} \int f_1 f \\ \int f_2 f \\ \vdots \\ \int f_n f \end{pmatrix}$$

This expression is very general. The only assumption we have used is the integrability of the partial products of the form $f_i f_j$ and $f_i f$. Since no special assumptions on the integral were used, it is also possible to use a discrete version of this equation. Assume that the function f has been defined at m points and let the symbol $\sum f_i f_j$ stand for $\sum_{k=1}^{m} f_i(x_k) f_j(x_k)$. In this case the above expression transforms to

$$\begin{pmatrix} \sum f_1 f_1 & \sum f_1 f_2 & \cdots & \sum f_1 f_n \\ \sum f_2 f_1 & \sum f_2 f_2 & & \sum f_2 f_n \\ \vdots & & \ddots & \vdots \\ \sum f_n f_1 & \sum f_n f_2 & \cdots & \sum f_n f_n \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} \sum f_1 f \\ \sum f_2 f \\ \vdots \\ \sum f_n f \end{pmatrix}$$

The general formula for the polynomial approximation of m data points $(x_1, y_1), \ldots, (x_m, y_m)$ using the primitive functions $x^0, x^1, x^2, \ldots, x^{n-1}$ translates directly into

$$\begin{pmatrix} m & \sum x_i & \cdots & \sum x_i^{n-1} \\ \sum x_i & \sum x_i^2 & & \sum x_i^n \\ \vdots & & \ddots & \vdots \\ \sum x_i^{n-1} & \sum x_i^n & \cdots & \sum x_i^{2n-2} \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} \sum y_i \\ \sum x_i y_i \\ \vdots \\ \sum x_i^{n-1} y_i \end{pmatrix}$$

In the case of base functions that are mutually orthogonal, the integrals $\int f_k f_j$ vanish when $k \neq j$. In this case the $n \times n$ matrix is diagonal and the above equation becomes very simple. Assume, as in the case of the Fourier transform, that the functions are sines and cosines of the form $\sin(k_i x)$ and $\cos(k_j x)$. Assume that no two sine functions have the same integer wave number k_i and no two cosine functions the same integer wave number k_j. In this case the integral $\int_0^{2\pi} \sin(k_i x) \sin(k_j x)$ is equal to π, whenever $i = j$, otherwise it vanishes. The same is true for the cosine functions. The expression transforms then to

$$\pi \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} \int f_1 f \\ \int f_2 f \\ \vdots \\ \int f_n f \end{pmatrix}$$

which is just another way of computing the coefficients for the Fourier approximation of the function f.

2.5.2 The Hadamard–Walsh transform

In our case we are interested in expressing Boolean functions in terms of a set of primitive functions. We adopt bipolar coding, so that now the logical value false is represented by -1 and the logical value true by 1. In the case of n logical variables $x_1, \ldots, x_n$ and the logical functions defined on them, we can use the following set of 2^n primitive functions:

- The constant function $(x_1, \ldots, x_n) \mapsto 1$
- The $\binom{n}{k}$ monomials $(x_1, \ldots, x_n) \mapsto x_{l_1} x_{l_2} \cdots x_{l_k}$, where $k = 1, \ldots, n$ and $l_1, l_2, \ldots, l_k$ is a set of k different indices in $\{1, 2, \ldots, n\}$

All these functions are mutually orthogonal. In the case of two input variables the transformation becomes:

$$4 \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & -1 & 1 \\ -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 \end{pmatrix} \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{pmatrix}$$

In the general case we compute 2^n coefficients using the formula

$$\begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_{2^n} \end{pmatrix} = H_n \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_{2^n} \end{pmatrix}$$

where the matrix H_n is defined recursively as

$$H_n = \frac{1}{2} \begin{pmatrix} H_{n-1} & H_{n-1} \\ -H_{n-1} & H_{n-1} \end{pmatrix}$$

whereby

$$H_1 = \frac{1}{2} \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix}.$$

The AND function can be expressed using this simple prescription as

$$x_1 \wedge x_2 = \frac{1}{4}(-2 + 2x_1 + 2x_2 + 2x_1 x_2).$$

The coefficients are the result of the following computation:

$$\begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix} = \frac{1}{4} \begin{pmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & -1 & 1 \\ -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 \end{pmatrix} \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{pmatrix}.$$

The expressions obtained for the logical functions can be wired as networks using weighted edges and only two operations, addition and binary multiplication. The Hadamard–Walsh transform is consequently a method for the synthesis of Boolean functions. The next step, that is, the optimization of the number of components, demands additional techniques which have been extensively studied in the field of combinatorics.

2.5.3 Applications of threshold logic

Threshold units can be used in any application in which we want to reduce the execution time of a logic operation to possibly just two layers of computational delay without employing a huge number of computing elements. It has been shown that the *parity* and *majority* functions, for example, cannot be implemented in a fixed number of layers of computation without using an exponentially growing number of conventional logic gates [148, 464], even when unbounded fan-in is used. The majority function k out of n is a threshold function implementable with just a single McCulloch–Pitts unit. Although circuits built from n threshold units can be built using a polynomial number $P(n)$ of conventional gates the main difference is that conventional circuits cannot guarantee a *constant* delay. With threshold elements we can build multiplication or division circuits that guarantee a constant delay for 32 or 64-bit operands. Any symmetric Boolean function of n bits can in fact be built from two layers of computing units using $n+1$ gates [407]. Some authors have developed circuits of threshold networks for fast multiplication and division, which are capable of operating with constant delay for a variable number of data bits [405]. Threshold logic offers thus the possibility of harnessing parallelism at the level of the basic arithmetic operations.

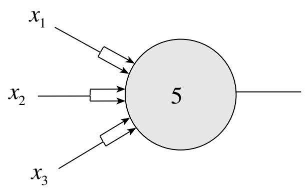


Fig. 2.26. Fault-tolerant gate

Threshold logic also offers a simpler way to achieve fault-tolerance. Figure 2.26 shows an example of a unit that can be used to compute the conjunction of three inputs with inherent fault tolerance. Assume that three inputs x_1, x_2, x_3 can be transmitted, each with probability p of error. The probability of a false result when x_1, x_2 and x_3 are equal, and we are computing the conjunction of the three inputs, is $3p$, since we assume that all three values are transmitted independently of each other. But assume that we transmit each value using two independent lines. The gate of Figure 2.26 has a threshold of 5, that is, it will produce the correct result even in the case where an input value is transmitted with an error. The probability that exactly two ones arrive as zeros is p^2 and, since there are 15 combinations of two out of six lines, the probability of getting the wrong answer is $15p^2$ in this case. If p is small enough then $15p^2 < 3p$ and the performance of the gate is improved for this combination of input values. Other combinations can be analyzed in a similar way. If threshold units are more reliable than the communication channels, redundancy can be exploited to increase the reliability of any computing system.

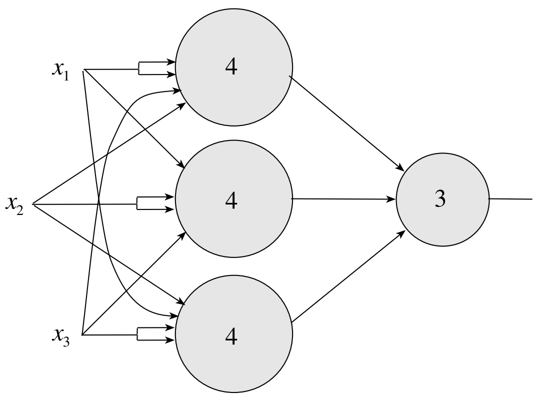


Fig. 2.27. A fault-tolerant AND built of noisy components

When the computing units are unreliable, fault tolerance is achieved using redundant networks. Figure 2.27 is an example of a network built using four units. Assume that the first three units connected directly to the three bits of input x_1, x_2, x_3 all fire with probability 1 when the total excitation is greater than or equal to the threshold θ but also with probability p when it is $\theta - 1$. The duplicated connections add redundancy to the transmitted bit, but in such a way that all three units fire with probability one when the three bits are 1. Each unit also fires with probability p if two out of three inputs are 1. However each unit reacts to a different combination. The last unit, finally, is also noisy and fires any time the three units in the first level fire and also with probability p when two of them fire. Since, in the first level, at most one unit fires when just two inputs are set to 1, the third unit will only fire when all three inputs are 1. This makes the logical circuit, the AND function of three inputs, built out of unreliable components error-proof. The network can be simplified using the approach illustrated in Figure 2.26.

2.6 Historical and bibliographical remarks

Warren McCulloch started pondering networks of artificial neurons as early as 1927 but had problems formulating a general, biologically plausible model since at that time inhibition in neurons had not yet been confirmed. He also had problems with the mathematical treatment of recurrent networks. Inhibition and recurrent pathways in the brain were confirmed in the 1930s and this cleared the way for McCulloch's investigations.

The McCulloch–Pitts model was proposed at a time when Norbert Wiener and Arturo Rosenblueth had started discussing the important role of feedback in biological systems and servomechanisms [301]. Wiener and his circle had published some of these ideas in 1943 [297]. Wiener's book *Cybernetics*, which was the first best-seller in the field of Artificial Intelligence, is the most influential work of this period. The word *cybernetics* was coined by Wiener and was intended to underline the importance of adaptive control in living and artificial systems. Wiener himself was a polymath, capable of doing first class research in mathematics as well as in other fields, such as physiology and medicine.

McCulloch and Pitts' model was superseded rapidly by more powerful approaches. Although threshold elements are, from the combinatorial point of view, more versatile than conventional logic gates, there is a problem with the assumed unlimited fan-in. Current technology has been optimized to handle a limited number of incoming signals into a gate. A possible way of circumventing the electrical difficulties could be the use of optical computing elements capable of providing unlimited fan-in [278]. Another alternative is the definition of a maximal fan-in for threshold elements that could be produced using conventional technology. Some experiments have been conducted in this direction and computers have been designed using exclusively threshold logic elements. The *DONUT* computer of Lewis and Coates was built in the 1960s using 614 gates with a maximal fan-in of 15. The same processor built with NOR gates with a maximal fan-in of 4 required 2127 gates, a factor of approximately 3.5 more components than in the former case [271].

John von Neumann [326] extensively discussed the model of McCulloch and Pitts and looked carefully at its fault tolerance properties. He examined the question of how to build a reliable computing mechanism built of unreliable components. However, he dealt mainly with the redundant coding of the units' outputs and a canonical building block, whereas McCulloch and his collaborator Manuel Blum later showed how to build reliable networks out of general noisy threshold elements [300]. Winograd and Cowan generalized this approach by replicating modules according to the requirements of an error-correcting code [458]. They showed that sparse coding of the input signals, coupled with error correction, makes possible fault-tolerant networks even in the presence of transmission or computational noise [94].

Exercises

1. Design a McCulloch–Pitts unit capable of recognizing the letter “T” digitized in a 10×10 array of pixels. Dark pixels should be coded as ones, white pixels as zeroes.
2. Build a recurrent network capable of adding two sequential streams of bits of arbitrary finite length.
3. Show that no finite automaton can compute the product of two sequential streams of bits of arbitrary finite length.
4. The parity of n given bits is 1 if an odd number of them is equal to 1, otherwise it is 0. Build a network of McCulloch–Pitts units capable of computing the parity function of two, three, and four given bits.
5. How many possible states can assume the binary scaler in Figure 2.21? Write the state and output tables for an equivalent finite automaton.
6. Design a network like the one shown in Figure 2.24 capable of simulating the finite automaton of the previous exercise.
7. Find polynomial expressions corresponding to the OR and XOR Boolean functions using the Hadamard–Walsh transform.
8. Show that the Hadamard–Walsh transform can be computed recursively, so that the number of multiplications becomes $O(n \log n)$, where n is the dimension of the vectors transformed (with n a power of two).
9. What is the probability of error in the case of the fault-tolerant gate shown in Figure 2.26? Consider one, two, and three faulty input bits.
10. The network in Figure 2.27 consists of unreliable computing units. Simplify the network. What happens if the units *and* the transmission channels are unreliable?

3

Weighted Networks – The Perceptron

3.1 Perceptrons and parallel processing

In the previous chapter we arrived at the conclusion that McCulloch–Pitts units can be used to build networks capable of computing any logical function and of simulating any finite automaton. From the biological point of view, however, the types of network that can be built are not very relevant. The computing units are too similar to conventional logic gates and the network must be completely specified before it can be used. There are no free parameters which could be adjusted to suit different problems. Learning can only be implemented by modifying the connection pattern of the network and the thresholds of the units, but this is necessarily more complex than just adjusting numerical parameters. For that reason, we turn our attention to weighted networks and consider their most relevant properties. In the last section of this chapter we show that simple weighted networks can provide a computational model for regular neuronal structures in the nervous system.

3.1.1 Perceptrons as weighted threshold elements

In 1958 Frank Rosenblatt, an American psychologist, proposed the *perceptron*, a more general computational model than McCulloch–Pitts units. The essential innovation was the introduction of numerical weights and a special interconnection pattern. In the original Rosenblatt model the computing units are threshold elements and the connectivity is determined stochastically. Learning takes place by adapting the weights of the network with a numerical algorithm. Rosenblatt's model was refined and perfected in the 1960s and its computational properties were carefully analyzed by Minsky and Papert [312]. In the following, Rosenblatt's model will be called the *classical perceptron* and the model analyzed by Minsky and Papert the *perceptron*.

The classical perceptron is in fact a whole network for the solution of certain pattern recognition problems. In Figure 3.1 a projection surface called the *retina* transmits binary values to a layer of computing units in the projection area. The connections from the retina to the projection units are deterministic and non-adaptive. The connections to the second layer of computing elements and from the second to the third are stochastically selected in order to make the model biologically plausible. The idea is to train the system to recognize certain input patterns in the connection region, which in turn leads to the appropriate path through the connections to the reaction layer. The learning algorithm must derive suitable weights for the connections.

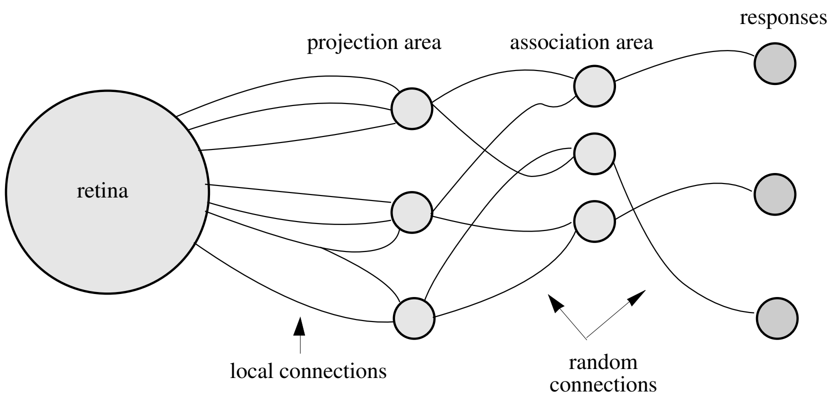


Fig. 3.1. The classical perceptron [after Rosenblatt 1958]

Rosenblatt's model can only be understood by first analyzing the elementary computing units. From a formal point of view, the only difference between McCulloch–Pitts elements and perceptrons is the presence of weights in the networks. Rosenblatt also studied models with some other differences, such as putting a limit on the maximum acceptable fan-in of the units.

Minsky and Papert distilled the essential features from Rosenblatt's model in order to study the computational capabilities of the perceptron under different assumptions. In the model used by these authors there is also a retina of pixels with binary values on which patterns are projected. Some pixels from the retina are directly connected to logic elements called predicates which can compute a single bit according to the input. Interestingly, these predicates can be as computationally complex as we like; for example, each predicate could be implemented using a supercomputer. There are some constraints however, such as the number of points in the retina that can be simultaneously examined by each predicate or the distance between those points. The predicates transmit their binary values to a weighted threshold element which is in charge of reaching the final decision in a pattern recognition problem. The question is then, what kind of patterns can be recognized in this massively parallel manner using a single threshold element at the output of the network? Are there limits to what we can compute in parallel using unlimited processing power for each predicate, when each predicate cannot itself look at the whole retina? The answer to this problem in some ways resembles the speedup problem in parallel processing, in which we ask what percentage of a computational task can be parallelized and what percentage is inherently sequential.

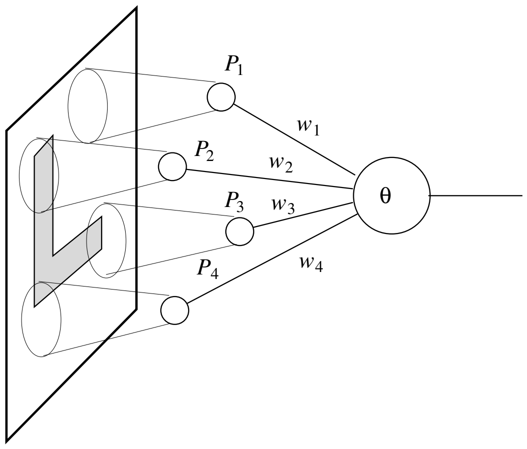


Fig. 3.2. Predicates and weights of a perceptron

Figure 3.2 illustrates the model discussed by Minsky and Papert. The predicates P_1 to P_4 deliver information about the points in the projection surface that comprise their *receptive fields*. The only restriction on the computational capabilities of the predicates is that they produce a binary value and the receptive field cannot cover the whole retina. The threshold element collects the outputs of the predicates through weighted edges and computes the final decision. The system consists in general of n predicates $P_1, P_2, \ldots, P_n$ and the corresponding weights $w_1, w_2, \ldots, w_n$. The system fires only when $\sum_{i=1}^{n} w_i P_i \geq \theta$, where θ is the threshold of the computing unit at the output.

3.1.2 Computational limits of the perceptron model

Minsky and Papert used their simplified perceptron model to investigate the computational capabilities of weighted networks. Early experiments with Rosenblatt's model had aroused unrealistic expectations in some quarters, and there was no clear understanding of the class of pattern recognition problems which it could solve efficiently. To explore this matter the number of predicates in the system is fixed, and although they possess unbounded computational power, the final bottleneck is the parallel computation with a single threshold element. This forces each processor to *cooperate* by producing a partial result pertinent to the global decision. The question now is which problems can be solved in this way and which cannot.

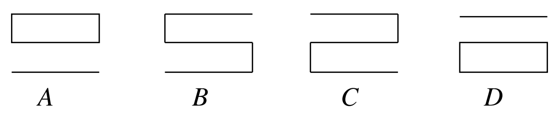
The system considered by Minsky and Papert at first appears to be a strong simplification of parallel decision processes, but it contains some of the most important elements differentiating between *sequential* and *parallel* processing. It is known that when some algorithms are parallelized, an irreducible sequential component sometimes limits the maximum achievable speedup. The mathematical relation between speedup and irreducible sequential portion of an algorithm is known as *Amdahl's law* [187]. In the model considered above the central question is, are there pattern recognition problems in which we are forced to analyze sequentially the output of the predicates associated with each receptive field or not? Minsky and Papert showed that problems of this kind do indeed exist which cannot be solved by a single perceptron acting as the last decision unit.

The limits imposed on the receptive fields of the predicates are based on realistic assumptions. The predicates are fixed in advance and the pattern recognition problem can be made arbitrarily large (by expanding the retina). According to the number of points and their connections to the predicates, Minsky and Papert differentiated between

- Diameter limited perceptrons: the receptive field of each predicate has a limited diameter.
- Perceptrons of limited order: each receptive field can only contain up to a certain maximum number of points.
- Stochastic perceptrons: each receptive field consists of a number of randomly chosen points

Some patterns are more difficult to identify than others and this structural classification of perceptrons is a first attempt at defining something like complexity classes for pattern recognition. Connectedness is an example of a property that cannot be recognized by constrained systems.

Proposition 6. *No diameter limited perceptron can decide whether a geometric figure is connected or not.*



Proof. We proceed by contradiction, assuming that a perceptron can decide whether a figure is connected or not. Consider the four patterns shown above; notice that only the middle two are connected.

Since the diameters of the receptive fields are limited, the patterns can be stretched horizontally in such a way that no single receptive field contains points from both the left and the right ends of the patterns. In this case we have three different groups of predicates: the first group consists of those predicates whose receptive fields contain points from the left side of a pattern. Predicates of the second group are those whose receptive fields cover the right side of a pattern. All other predicates belong to the third group. In Figure 3.3 the receptive fields of the predicates are represented by circles.

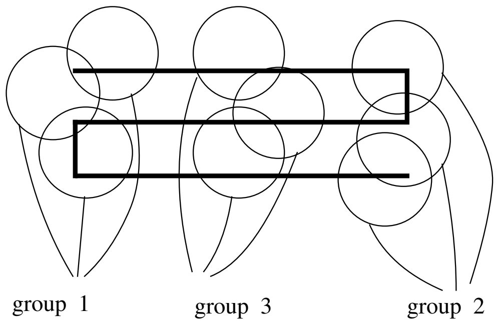


Fig. 3.3. Receptive fields of predicates

All predicates are connected to a threshold element through weighted edges which we denote by the letter w with an index. The threshold element decides whether a figure is connected or not by performing the computation

$$S = \sum_{P_i \in \text{group } 1} w_{1i} P_i + \sum_{P_i \in \text{group } 2} w_{2i} P_i + \sum_{P_i \in \text{group } 3} w_{3i} P_i - \theta \geq 0.$$

If S is positive the figure is recognized as connected, as is the case, for example, in Figure 3.3.

If the disconnected pattern A is analyzed, then we should have $S < 0$. Pattern A can be transformed into pattern B without affecting the output of the predicates of group 3, which do not recognize the difference since their receptive fields do not cover the sides of the figures. The predicates of group 2 adjust their outputs by $\Delta_2 S$ so that now

$$S + \Delta_2 S \geq 0 \Rightarrow \Delta_2 S \geq -S.$$

If pattern A is transformed into pattern C, the predicates of group 1 adjust their outputs so that the threshold element receives a net excitation, i.e.,

$$S + \Delta_1 S \geq 0 \Rightarrow \Delta_1 S \geq -S.$$

However, if pattern A is transformed into pattern D, the predicates of group 1 cannot distinguish this case from the one for figure C and the predicates of group 2 cannot distinguish this case from the one for figure B. Since the predicates of group 3 do not change their output we have

$$\Delta S = \Delta_2 S + \Delta_1 S \geq -2S,$$

and from this

$$S + \Delta S \geq -S > 0.$$

The value of the new sum can only be positive and the whole system classifies figure D as connected. Since this is a contradiction, such a system cannot exist. □

Proposition 6 states only that the connectedness of a figure is a global property which cannot be decided locally. If no predicate has access to the whole figure, then the only alternative is to process the outputs of the predicates sequentially.

There are some other difficult problems for perceptrons. They cannot decide, for example, whether a set of points contains an even or an odd number of elements when the receptive fields cover only a limited number of points.

3.2 Implementation of logical functions

In the previous chapter we discussed the synthesis of Boolean functions using McCulloch–Pitts networks. Weighted networks can achieve the same results with fewer threshold gates, but the issue now is which functions can be implemented using a single unit.

3.2.1 Geometric interpretation

In each of the previous sections a threshold element was associated with a whole set of predicates or a network of computing elements. From now on, we will deal with perceptrons as isolated threshold elements which compute their output without delay.

Definition 1. *A simple perceptron is a computing unit with threshold θ which, when receiving the n real inputs $x_1, x_2, \ldots, x_n$ through edges with the associated weights $w_1, w_2, \ldots, w_n$, outputs 1 if the inequality $\sum_{i=1}^{n} w_i x_i \geq \theta$ holds and otherwise 0.*

The origin of the inputs is not important, whether they come from other perceptrons or another class of computing units. The geometric interpretation of the processing performed by perceptrons is the same as with McCulloch–Pitts elements. A perceptron separates the input space into two half-spaces. For points belonging to one half-space the result of the computation is 0, for points belonging to the other it is 1.

Figure 3.4 shows this for the case of two variables x_1 and x_2. A perceptron with threshold 1, at which two edges with weights 0.9 and 2.0 impinge, tests the condition

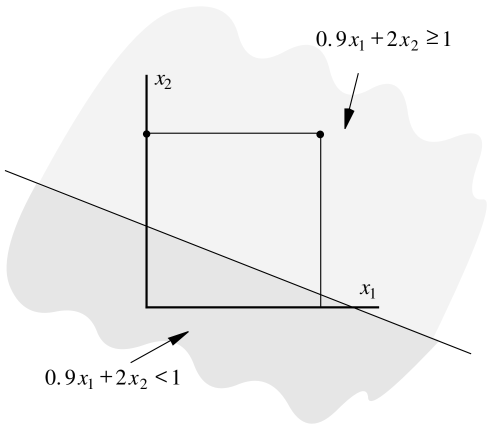


Fig. 3.4. Separation of input space with a perceptron

$$0.9x_1 + 2x_2 \geq 1.$$

It is possible to generate arbitrary separations of input space by adjusting the parameters of this example.

In many cases it is more convenient to deal with perceptrons of threshold zero only. This corresponds to linear separations which are forced to go through the origin of the input space. The two perceptrons in Figure 3.5 are equivalent. The threshold of the perceptron to the left has been converted into the weight $-\theta$ of an additional input channel connected to the constant 1. This extra weight connected to a constant is called the *bias* of the element.

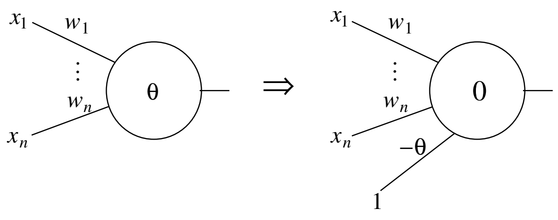


Fig. 3.5. A perceptron with a bias

Most learning algorithms can be stated more concisely by transforming thresholds into biases. The input vector $(x_1, x_2, \ldots, x_n)$ must be extended with an additional 1 and the resulting $(n+1)$-dimensional vector $(x_1, x_2, \ldots, x_n, 1)$ is called the *extended input vector*. The extended weight vector associated with this perceptron is $(w_1, \ldots, w_n, w_{n+1})$, whereby $w_{n+1} = -\theta$.

3.2.2 The XOR problem

We can now deal with the problem of determining which logical functions can be implemented with a single perceptron. A perceptron network is capable of computing any logical function, since perceptrons are even more powerful than unweighted McCulloch–Pitts elements. If we reduce the network to a single element, which functions are still computable?

Taking the functions of two variables as an example we can gain some insight into this problem. Table 3.1 shows all 16 possible Boolean functions of two variables f_0 to f_{15}. Each column f_i shows the value of the function for each combination of the two variables x_1 and x_2. The function f_0, for example, is the zero function whereas f_{14} is the OR-function.

Table 3.1. The 16 Boolean functions of two variables

x_1	x_2	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Perceptron-computable functions are those for which the points whose function value is 0 can be separated from the points whose function value is 1 using a line. Figure 3.6 shows two possible separations to compute the OR and the AND functions.

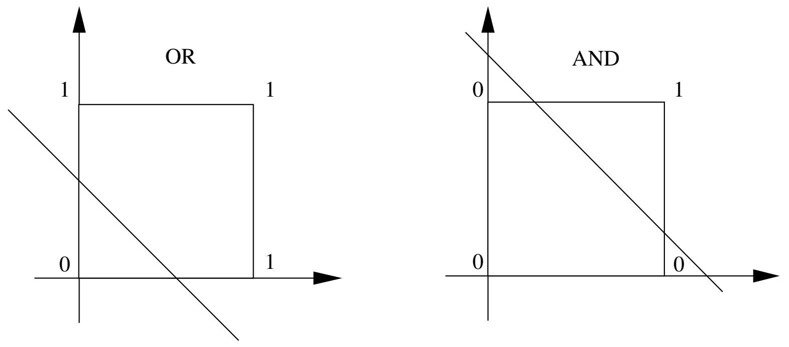


Fig. 3.6. Separations of input space corresponding to OR and AND

It is clear that two of the functions in the table cannot be computed in this way. They are the function XOR and identity (f_6 and f_9). It is intuitively evident that no line can produce the necessary separation of the input space. This can also be shown analytically.

Let w_1 and w_2 be the weights of a perceptron with two inputs, and θ its threshold. If the perceptron computes the XOR function the following four inequalities must be fulfilled:

$$\begin{array}{llll} x_1 = 0 \; x_2 = 0 & w_1x_1 + w_2x_2 = 0 & \Rightarrow & 0 < \theta \\ x_1 = 1 \; x_2 = 0 & w_1x_1 + w_2x_2 = w_1 & \Rightarrow & w_1 \geq \theta \\ x_1 = 0 \; x_2 = 1 & w_1x_1 + w_2x_2 = w_2 & \Rightarrow & w_2 \geq \theta \\ x_1 = 1 \; x_2 = 1 & w_1x_1 + w_2x_2 = w_1 + w_2 & \Rightarrow & w_1 + w_2 < \theta \end{array}$$

Since θ is positive, according to the first inequality, w_1 and w_2 are positive too, according to the second and third inequalities. Therefore the inequality $w_1 + w_2 < \theta$ cannot be true. This contradiction implies that no perceptron capable of computing the XOR function exists. An analogous proof holds for the function f_9.

3.3 Linearly separable functions

The example of the logical functions of two variables shows that the problem of perceptron computability must be discussed in more detail. In this section we provide the necessary tools to deal more effectively with functions of n arguments.

3.3.1 Linear separability

We can deduce from our experience with the XOR function that many other logical functions of several arguments must exist which cannot be computed with a threshold element. This fact has to do with the geometry of the n-dimensional hypercube whose vertices represent the combination of logic values of the arguments. Each logical function separates the vertices into two classes. If the points whose function value is 1 cannot be separated with a linear cut from the points whose function value is 0, the function is not perceptron-computable. The following two definitions give this problem a more general setting.

Definition 2. *Two sets of points A and B in an n-dimensional space are called linearly separable if $n+1$ real numbers $w_1, \ldots, w_{n+1}$ exist, such that every point $(x_1, x_2, \ldots, x_n) \in A$ satisfies $\sum_{i=1}^{n} w_i x_i \geq w_{n+1}$ and every point $(x_1, x_2, \ldots, x_n) \in B$ satisfies $\sum_{i=1}^{n} w_i x_i < w_{n+1}$*

Since a perceptron can only compute linearly separable functions, an interesting question is how many linearly separable functions of n binary arguments there are. When $n = 2$, 14 out of the 16 possible Boolean functions are linearly separable. When $n = 3$, 104 out of 256 and when $n = 4$, 1882 out of 65536 possible functions are linearly separable. Although there has been extensive research on linearly separable functions in recent years, no formula for expressing the number of linearly separable functions as a function of n has yet been found. However we will provide some upper bounds for this number in the following chapters.

3.3.2 Duality of input space and weight space

The computation performed by a perceptron can be visualized as a linear separation of input space. However, when trying to find the appropriate weights for a perceptron, the search process can be better visualized in weight space. When m real weights must be determined, the search space is the whole of $\mathbb{R}^m$.

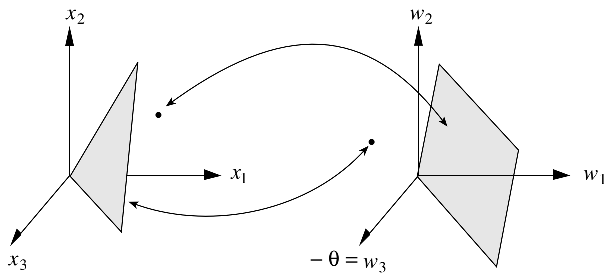


Fig. 3.7. Illustration of the duality of input and weight space

For a perceptron with n input lines, finding the appropriate linear separation amounts to finding $n+1$ free parameters (n weights and the bias). These $n+1$ parameters represent a point in $(n+1)$-dimensional weight space. Each time we pick one point in weight space we are choosing one combination of weights and a specific linear separation of input space. This means that every point in $(n+1)$-dimensional weight space can be associated with a hyperplane in $(n+1)$-dimensional extended input space. Figure 3.7 shows an example. Each combination of three weights, w_1, w_2, w_3, which represent a point in weight space, defines a separation of input space with the plane $w_1x_1 + w_2x_2 + w_3x_3 = 0$.

There is the same kind of relation in the inverse direction, from input to weight space. If we want the point x_1, x_2, x_3 to be located in the positive half-space defined by a plane, we need to determine the appropriate weights w_1, w_2 and w_3. The inequality

$$w_1x_1 + w_2x_2 + w_3x_3 \geq 0$$

must hold. However this inequality defines a linear separation of weight space, that is, the point (x_1, x_2, x_3) defines a cutting plane in weight space. Points in one space are mapped to planes in the other and vice versa. This complementary relation is called *duality.* Input and weight space are dual spaces and we can visualize the computations done by perceptrons and learning algorithms in any one of them. We will switch from one visualization to the other as necessary or convenient.

3.3.3 The error function in weight space

Given two sets of patterns which must be separated by a perceptron, a learning algorithm should automatically find the weights and threshold necessary for the solution of the problem. The *perceptron learning algorithm* can accomplish this for threshold units. Although proposed by Rosenblatt it was already known in another context [10].

Assume that the set A of input vectors in n-dimensional space must be separated from the set B of input vectors in such a way that a perceptron computes the binary function f_w with $f_w(x) = 1$ for $x \in A$ and $f_w(x) = 0$ for $x \in B$. The binary function f_w depends on the set w of weights and threshold. The *error function* is the number of false classifications obtained using the weight vector w. It can be defined as:

$$E(w) = \sum_{x \in A} (1 - f_w(x)) + \sum_{x \in B} f_w(x).$$

This is a function defined over all of weight space and the aim of perceptron learning is to minimize it. Since $E(w)$ is positive or zero, we want to reach the global minimum where $E(w) = 0$. This will be done by starting with a random weight vector w, and then searching in weight space a better alternative, in an attempt to reduce the error function $E(w)$ at each step.

3.3.4 General decision curves

A perceptron makes a decision based on a linear separation of the input space. This reduces the kinds of problem solvable with a single perceptron. More general separations of input space can help to deal with other kinds of problem unsolvable with a single threshold unit. Assume that a single computing unit can produce the separation shown in Figure 3.8. Such a separation of the input space into two regions would allow the computation of the XOR function with a single unit. Functions used to discriminate between regions of input space are called *decision curves* [329]. Some of the decision curves which have been studied are polynomials and splines.

In statistical pattern recognition problems we assume that the patterns to be recognized are grouped in clusters in input space. Using a combination of decision curves we try to isolate one cluster from the others. One alternative is combining several perceptrons to isolate a convex region of space. Other alternatives which have been studied are, for example, so-called Sigma-Pi units which, for a given input $x_1, x_2, \ldots, x_n$, compute the sum of all or some partial products of the form $x_i x_j$ [384].

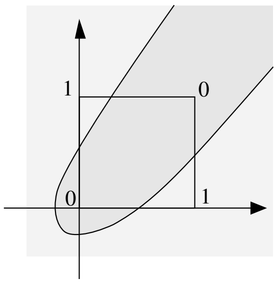


Fig. 3.8. Non-linear separation of input space

In the general case we want to distinguish between regions of space. A neural network must learn to identify these regions and to associate them with the correct response. The main problem is determining whether the free parameters of these decision regions can be found using a learning algorithm. In the next chapter we show that it is always possible to find these free parameters for linear decision curves, if the patterns to be classified are indeed linearly separable. Finding learning algorithms for other kinds of decision curves is an important research topic not dealt with here [45, 4].

3.4 Applications and biological analogy

The appeal of the perceptron model is grounded on its simplicity and the wide range of applications that it has found. As we show in this section, weighted threshold elements can play an important role in image processing and computer vision.

3.4.1 Edge detection with perceptrons

A good example of the pattern recognition capabilities of perceptrons is edge detection (Figure 3.9). Assume that a method of extracting the edges of a figure darker than the background (or the converse) is needed. Each pixel in the figure is compared to its immediate neighbors and in the case where the pixel is black and one of its neighbors white, it will be classified as part of an edge. This can be programmed sequentially in a computer, but since the decision about each point uses only local information, it is straightforward to implement the strategy in parallel.

Assume that the figures to be processed are projected on a screen in which each pixel is connected to a perceptron, which also receives inputs from its immediate neighbors. Figure 3.10 shows the shape of the receptive field (a so-called Moore neighborhood) and the weights of the connections to the perceptron. The central point is weighted with 8 and the rest with -1. In the field of image processing this is called a *convolution operator*, because it is

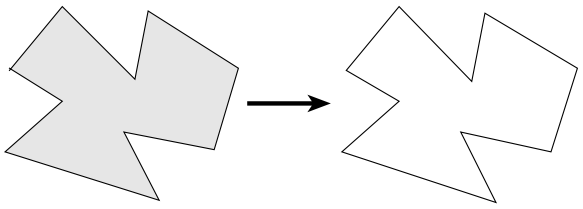


Fig. 3.9. Example of edge detection

used by centering it at each pixel of the image to produce a certain output value for each pixel. The operator shown has a maximum at the center of the receptive field and local minima at the periphery.

-1	-1	-1
-1	8	-1
-1	-1	-1

Fig. 3.10. Edge detection operator

Figure 3.11 shows the kind of interconnection we have in mind. A perceptron is needed for each pixel. The interconnection pattern repeats for each pixel in the projection lattice, taking care to treat the borders of the screen differently. The weights are those given by the edge detection operator.

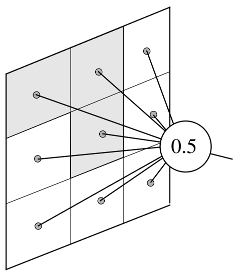


Fig. 3.11. Connection of a perceptron to the projection grid

For each pattern projected onto the screen, the weighted input is compared to the threshold 0.5. When all points in the neighborhood are black or all white, the total excitation is 0. In the situation shown below the total excitation is 5 and the point in the middle belongs to an edge.

There are many other operators for different uses, such as detecting horizontal or vertical lines or blurring or making a picture sharper. The size of the neighborhood can be adjusted to the specific application. For example, the operator

$$\begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix}$$

can be used to detect the vertical edges between a white surface to the left and a dark one to the right.

3.4.2 The structure of the retina

The visual pathway is the part of the human brain which is best understood. The retina can be conceived as a continuation of this organ, since it consists of neural cells organized in layers and capable of providing *in situ* some of the information processing necessary for vision. In frogs and other small vertebrates some neurons have been found directly in the retina which actually fire in the presence of a small blob in the visual field. These are bug detectors which tell these animals when a small insect has been sighted.

Researchers have found that the cells in the retina are interconnected in such a way that each nerve going from the eyes to the brain encodes a summary of the information detected by several photoreceptors in the retina. As in the case of the convolution operators discussed previously, each nerve transmits a signal which depends on the relative luminosity of a point in relation to its immediate neighborhood.

Figure 3.12 shows the interconnection pattern found in the retina [205, 111]. The cones and rods are the photoreceptors which react to photons by depolarizing. Horizontal cells compute the average luminosity in a region by connecting to the cones or rods in this region. Bipolar and ganglion cells fire only when the difference in the luminosity of a point is significantly higher than the average light intensity.

Although not all details of the retinal circuits have been reverse-engineered, there is a recurrent feature: each receptor cell in the retina is part of a roughly circular receptive field. The receptive fields of neighboring cells overlap. Their function is similar to the edge processing operators, because the neighborhood inhibits a neuron whereas a photoreceptor excites it, or conversely. This kind of weighting of the input has a strong resemblance to the so-called Mexican hat function.

David Marr tried to summarize what we know about the visual pathway in humans and proposed his idea of a process in three stages, in which the brain first decomposes an image into features (edges, blobs, etc.), which are then used to build an interpretation of surfaces, depth relations and groupings of tokens (the "$2\frac{1}{2}$ sketch") and which in turn leads to a full interpretation of the objects present in the visual field (the primal sketch) [293]. He tried to explain the structure of the retina from the point of view of the computational machinery needed for vision. He proposed that at a certain stage of

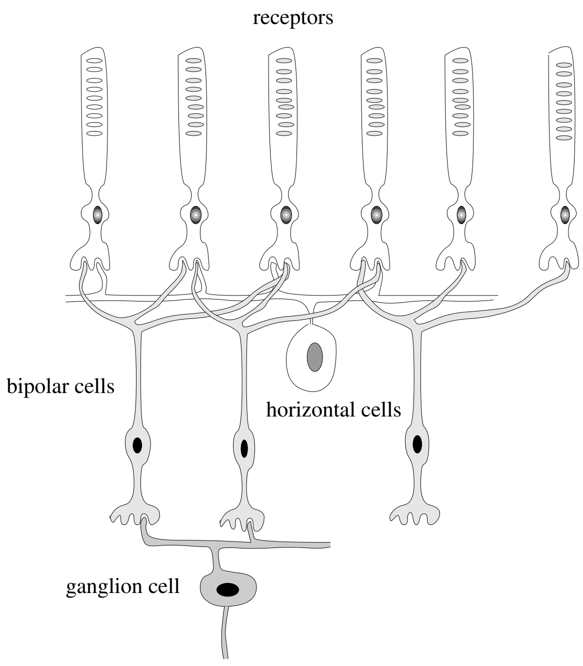
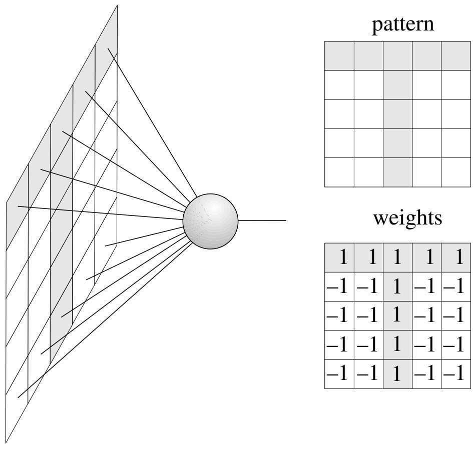


Fig. 3.12. The interconnection pattern in the retina



1	1	1	1	1
–1	–1	1	–1	–1
–1	–1	1	–1	–1
–1	–1	1	–1	–1
–1	–1	1	–1	–1

Fig. 3.13. Feature detector for the pattern T

the computation the retina blurs an image and then extracts from it contrast information. Blurring an image can be done by averaging at each pixel the values of this pixel and its neighbors. A Gaussian distribution of weights can be used for this purpose. Information about changes in darkness levels can be extracted using the sum of the second derivatives of the illumination function, the so-called Laplace operator $\nabla^2 = \partial^2/\partial x^2 + \partial^2/\partial y^2$. The composition of the Laplace operator and a Gaussian blurring corresponds to the Mexican hat function. Processing of the image is done by computing the convolution of the discrete version of the operator with the image. The 3×3 discrete version of this operator is the edge detection operator which we used before. Different levels of blurring, and thus feature extraction at several different resolution levels, can be controlled by adjusting the size of the receptive fields of the computing units. It seems that the human visual pathway also exploits feature detection at several resolution levels, which has led in turn to the idea of using several resolution layers for the computational analysis of images.

3.4.3 Pyramidal networks and the neocognitron

Single perceptrons can be thought of as *feature detectors*. Take the case of Figure 3.13 in which a perceptron is defined with weights adequate for recognizing the letter ‘T’ in which t pixels are black. If another ‘T’ is presented, in which one black pixel is missing, the excitation of the perceptron is $t - 1$. The same happens if one white pixel is transformed into a black one due to noise, since the weights of the connections going from points that should be white are -1. If the threshold of the perceptron is set to $t - 1$, then this perceptron will be capable of correctly classifying patterns with one noisy pixel. By adjusting the threshold of the unit, 2, 3 or more noisy pixels can be tolerated. Perceptrons thus compute the similarity of a pattern to the ideal pattern they have been designed to identify, and the threshold is the minimal similarity that we require from the pattern. Note that since the weights of the perceptron are correlated with the pattern it recognizes, a simple way to visualize the connections of a perceptron is to draw the pattern it identifies in its receptive field. This technique will be used below.

The problem with this pattern recognition scheme is that it only works if the patterns have been normalized in some way, that is, if they have been centered in the window to which the perceptron connects and their size does not differ appreciably from the ideal pattern. Also, any kind of translational shift will lead to ideal patterns no longer being recognized. The same happens in the case of rotations.

An alternative way of handling this problem is to try to detect patterns not in a single step, but in several stages. If we are trying, for example, to recognize handwritten digits, then we could attempt to find some small distinctive features such as lines in certain orientations and then combine our knowledge about the presence or absence of these features in a final logical decision. We should try to recognize these small features, regardless of their position on the projection screen.

The *cognitron* and *neocognitron* were designed by Fukushima and his colleagues as an attempt to deal with this problem and in some way to try to mimic the structure of the human vision pathway [144, 145]. The main idea of

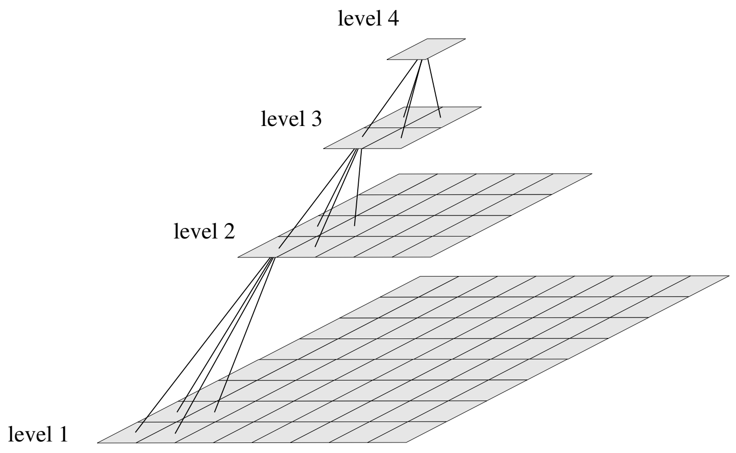


Fig. 3.14. Pyramidal architecture for image processing

the neocognitron is to transform the contents of the screen into other screens in which some features have been enhanced, and then again into other screens, and so on, until a final decision is made. The resolution of the screen can be changed from transformation to transformation or more screens can be introduced, but the objective is to reduce the representation to make a final classification in the last stage based on just a few points of input.

The general structure of the neural system proposed by Fukushima is a kind of variant of what is known in the image processing community as a pyramidal architecture, in which the resolution of the image is reduced by a certain factor from plane to plane [56]. Figure 3.14 shows an example of a quad-pyramid, that is, a system in which the resolution is reduced by a factor of four from plane to plane. Each pixel in one of the upper planes connects to four pixels in the plane immediately below and deals with them as elements of its receptive field. The computation to determine the value of the upper pixel can be arbitrary, but in our case we are interested in threshold computations. Note that in this case receptive fields do not overlap. Such architectures have been studied intensively to determine their capabilities as data structures for parallel algorithms [80, 57].

The neocognitron has a more complex architecture [280]. The image is transformed from the original plane to other planes to look for specific features.

Figure 3.15 shows the general strategy adopted in the neocognitron. The projection screen is transformed, deciding for each pixel if it should be kept white or black. This can be done by identifying the patterns shown for each of the three transformations by looking at each pixel and its eight neighbors. In the case of the first transformed screen only horizontal lines are kept; in the second screen only vertical lines and in the third screen only diagonal lines. The convolution operators needed have the same distribution of positive

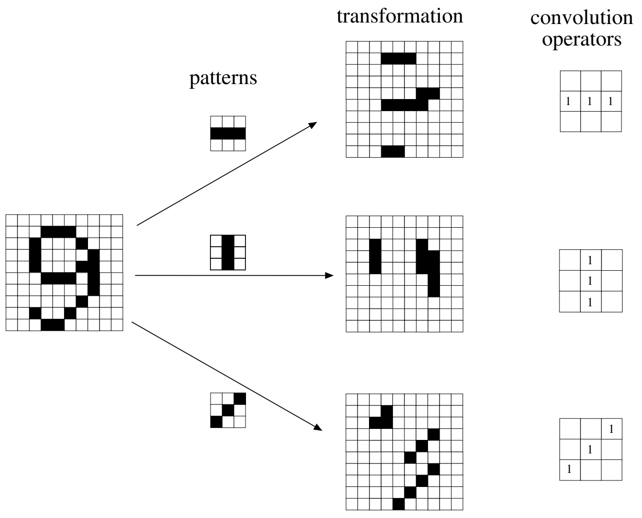


Fig. 3.15. Feature extraction in the neocognitron

weights, as shown for each screen. The rest of the weights is 0. Note that these special weights work better if the pattern has been previously 'skeletonized'.

Strictly speaking Fukushima's neocognitron uses linear computing units and not perceptrons. The units compute their total weighted input and this is interpreted as a kind of correlation index with the patterns that each unit can identify. This means that black and white patterns are transformed into patterns with shadings of gray, according to the output of each mapping unit. Figure 3.16 shows the general structure of the neocognitron network. The input layer of the network is called UC^0. This input layer is processed and converted into twelve different images numbered US^1_0 to US^1_{11} with the same resolution. The superindex in front of a name is the layer number and the subindex the number of the plane in this layer. The operators used to transform from UC^0 to each of the US^1_i planes have a receptive field of 3×3 pixels and one operator is associated with each pixel in the US^1_i planes. In each plane only one kind of feature is recognized. The first plane US^1_1, for example, could contain all the vertical edges found in UC^0, the second plane US^1_2 only diagonal edges, and so forth. The next level of processing is represented by the UC^1_j planes. Each pixel in one of these planes connects to a receptive field in one or two of the underlying US^1_i planes. The weights are purely excitatory and the effect of this layer is to overlap the activations of the selected US^1_i images, blurring them at the same time, that is, making the patterns wider. This is achieved by transforming each pixel's value in the weighted average of its own and its neighbor's values.

In the next level of processing each pixel in a US^2_i plane connects to a receptive field at the same position in all of the UC^1_j images. At this level the resolution of the US^2_i planes can be reduced, as in standard pyramidal

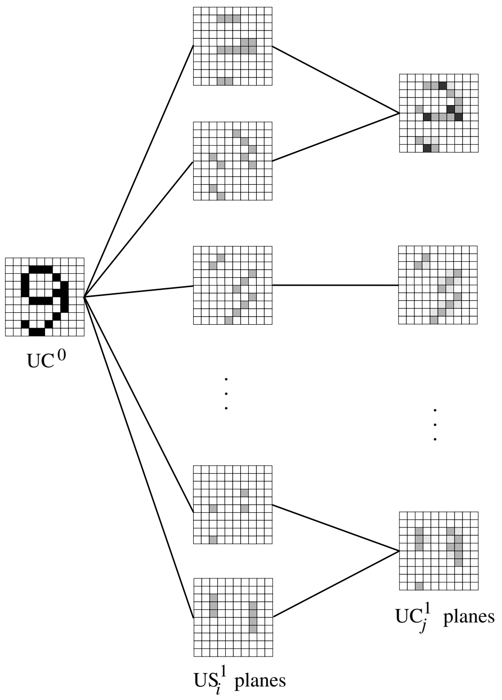


Fig. 3.16. The architecture of the neocognitron

architectures. Fig 3.16 shows the sizes of the planes used by Fukushima for handwritten digit recognition. Several layers of alternating US and UC planes are arranged in this way until at the plane UC^4 a classification of the handwritten digit in one of the classes $0, \ldots, 9$ is made. Finding the appropriate weights for the classification task is something we discuss in the next chapter. Fukushima has proposed several improvements of the original model [147] over the years.

The main advantage of the neocognitron as a pattern recognition device should be its tolerance to shifts and distortions. Since the UC layers blur the image and the US layers look for specific features, a certain amount of displacement or rotation of lines should be tolerated. This can happen, but the system is highly sensitive to the training method used and does not outperform other simpler neural networks [280]. Other authors have examined variations of the neocognitron which are more similar to pyramidal networks [463]. The neocognitron is just an example of a class of network which relies extensively on convolution operators and pattern recognition in small receptive fields. For an extensive discussion of the neocognitron consult [133].

3.4.4 The silicon retina

Carver Mead's group at Caltech has been active for several years in the field of *neuromorphic engineering*, that is, the production of chips capable of emulating the sensory response of some human organs. Their *silicon retina*, in particular, is able to simulate some of the features of the human retina.

Mead and his collaborators modeled the first three layers of the retina: the photoreceptors, the horizontal, and the bipolar cells [303, 283]. The horizontal cells are simulated in the silicon retina as a grid of resistors. Each photoreceptor (the dark points in Figure 3.17) is connected to each of its six neighbors and produces a potential difference proportional to the logarithm of the luminosity measured. The grid of resistors reaches electric equilibrium when an average potential difference has settled in. The individual neurons of the silicon retina fire only when the difference between the average and their own potential reaches a certain threshold.

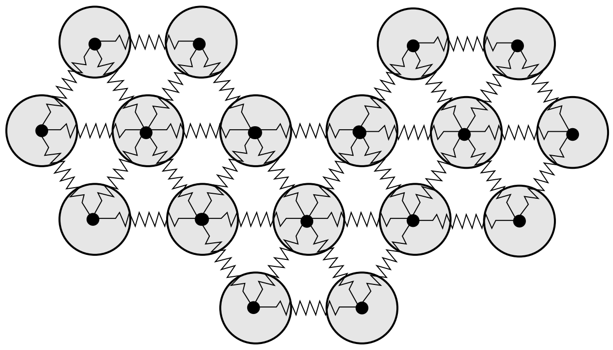


Fig. 3.17. Diagram of a portion of the silicon retina

The average potential S of n potentials S_i can be computed by letting each potential S_i be proportional to the logarithm of the measured light intensity H_i, that is,

$$S = \frac{1}{n}\sum_{i=1}^{n} S_i = \frac{1}{n}\sum_{i=1}^{n} \log H_i.$$

This expression can be transformed into

$$S = \frac{1}{n}\log(H_1 H_2 \cdots H_n) = \log(H_1 H_2 \cdots H_n)^{1/n}.$$

The equation tells us that the average potential is the logarithm of the geometric mean of the light intensities. A unit only fires when the measured intensity S_i minus the average intensity lies above a certain threshold γ, that is,

$$\log(H_i) - \log(H_1 H_2 \cdots H_n)^{1/n} \geq \gamma,$$

and this is valid only when

$$\log \frac{H_i}{(H_1 H_2 \cdots H_n)^{1/n}} \geq \gamma.$$

The units in the silicon retina fire when the relative luminosity of a point with respect to the background is significantly higher, such as in a human retina. We know from optical measurements that when outside on a sunny day, the black letters in a book reflect more photons on our eyes than white paper does in a room. Our eyes adjust automatically to compensate for the luminosity of the background so that we can recognize patterns and read books inside and outside.

3.5 Historical and bibliographical remarks

The perceptron was the first neural network to be produced commercially, although the first prototypes were used mainly in research laboratories. Frank Rosenblatt used the perceptron to solve some image recognition problems [185]. Some researchers consider the perceptron as the first serious abstract model of nervous cells [60].

It was not a coincidence that Rosenblatt conceived his model at the end of the 1950s. It was precisely in this period that researchers like Hubel and Wiesel were able to "decode" the structure of the retina and examine the structure of the receptive fields of neurons. At the beginning of the 1970s, researchers had a fair global picture of the architecture of the human eye [205]. David Marr's influential book *Vision* offered the first integrated picture of the visual system in a way that fused biology and engineering, by looking at the way the visual pathway actually computed partial results to be integrated in the raw visual sketch.

The book *Perceptrons* by Minsky and Papert was very influential among the AI community and is said to have affected the strategic funding decisions of research agencies. This book is one of the best ever written on its subject and set higher standards for neural network research, although it has been criticized for stressing the incomputability, not the computability results. The Dreyfus brothers [114] consider the reaction to *Perceptrons* as one of the milestones in the permanent conflict between the *symbolic* and the *connectionist* schools of thought in AI. According to them, reaction to the book opened the way for a long period of dominance of the symbolic approach. Minsky, for his part, now propagates an alternative massively parallel paradigm of a society of agents of consciousness which he calls a *society of mind* [313].

Convolution operators for image processing have been used for many years and are standard methods in the fields of image processing and computer vision. Chips integrating this kind of processing, like the silicon retina, have been produced in several variants and will be used in future robots. Some researchers dream of using similar chips to restore limited vision to blind persons with intact visual nerves, although this is, of course, still an extremely ambitious objective [123].

Exercises

1. Write a computer program that counts the number of linearly separable Boolean functions of 2, 3, and 4 arguments. *Hint*: Generate the perceptron weights randomly.
2. Consider a simple perceptron with n bipolar inputs and threshold $\theta = 0$. Restrict each of the weights to have the value -1 or 1. Give the smallest upper bound you can find for the number of functions from $\{-1, 1\}^n$ to $\{-1, 1\}$ which are computable by this perceptron [219]. Prove that the upper bound is sharp, i.e., that all functions are different.
3. Show that two finite linearly separable sets A and B can be separated by a perceptron with rational weights. Note that in Def. 2 the weights are real numbers.
4. Prove that the parity function of $n > 2$ binary inputs $x_1, x_2, \ldots, x_n$ cannot be computed by a perceptron.
5. Implement edge detection with a computer program capable of processing a computer image.
6. Write a computer program capable of simulating the silicon retina. Show the output produced by different pictures on the computer's screen.

4

Perceptron Learning

4.1 Learning algorithms for neural networks

In the two preceding chapters we discussed two closely related models, McCulloch–Pitts units and perceptrons, but the question of how to find the parameters adequate for a given task was left open. If two sets of points have to be separated linearly with a perceptron, adequate weights for the computing unit must be found. The operators that we used in the preceding chapter, for example for edge detection, used hand customized weights. Now we would like to find those parameters automatically. The *perceptron learning algorithm* deals with this problem.

A learning algorithm is an adaptive method by which a network of computing units self-organizes to implement the desired behavior. This is done in some learning algorithms by presenting some examples of the desired input-output mapping to the network. A correction step is executed iteratively until the network learns to produce the desired response. The learning algorithm is a closed loop of presentation of examples and of corrections to the network parameters, as shown in Figure 4.1.

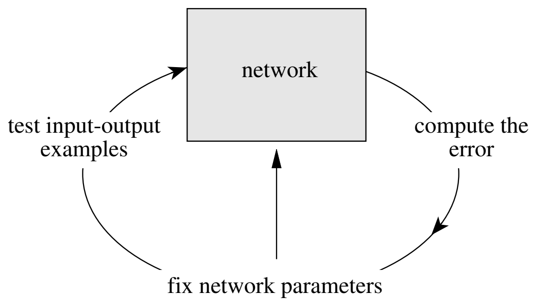


Fig. 4.1. Learning process in a parametric system

In some simple cases the weights for the computing units can be found through a sequential test of stochastically generated numerical combinations. However, such algorithms which look blindly for a solution do not qualify as "learning". A learning algorithm must adapt the network parameters according to previous experience until a solution is found, if it exists.

4.1.1 Classes of learning algorithms

Learning algorithms can be divided into *supervised* and *unsupervised* methods. Supervised learning denotes a method in which some input vectors are collected and presented to the network. The output computed by the network is observed and the deviation from the expected answer is measured. The weights are corrected according to the magnitude of the error in the way defined by the learning algorithm. This kind of learning is also called *learning with a teacher*, since a control process knows the correct answer for the set of selected input vectors.

Unsupervised learning is used when, for a given input, the exact numerical output a network should produce is unknown. Assume, for example, that some points in two-dimensional space are to be classified into three clusters. For this task we can use a classifier network with three output lines, one for each class (Figure 4.2). Each of the three computing units at the output must specialize by firing only for inputs corresponding to elements of each cluster. If one unit fires, the others must keep silent. In this case we do not know a priori which unit is going to specialize on which cluster. Generally we do not even know how many well-defined clusters are present. Since no "teacher" is available, the network must organize itself in order to be able to associate clusters with units.

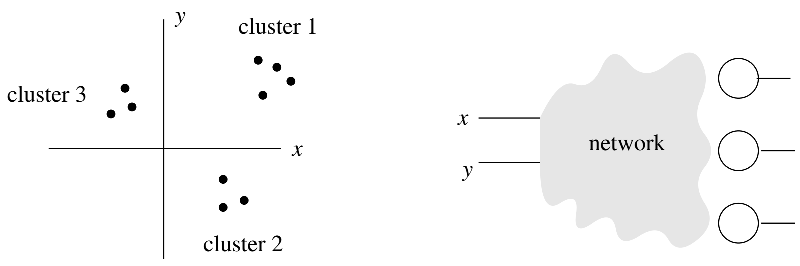


Fig. 4.2. Three clusters and a classifier network

Supervised learning is further divided into methods which use reinforcement or error correction. Reinforcement learning is used when after each presentation of an input-output example we only know whether the network produces the desired result or not. The weights are updated based on this information (that is, the Boolean values *true* or *false*) so that only the input vector can be used for weight correction. In learning with error correction, the magnitude of the error, together with the input vector, determines the magnitude of the corrections to the weights, and in many cases we try to eliminate the error in a single correction step.

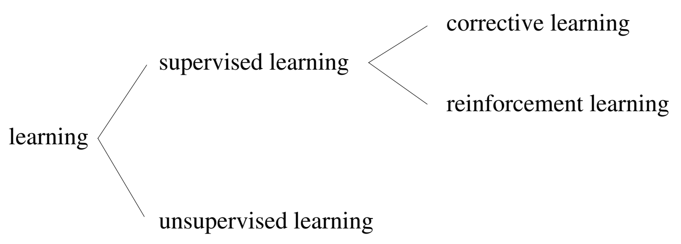


Fig. 4.3. Classes of learning algorithms

The perceptron learning algorithm is an example of supervised learning with reinforcement. Some of its variants use supervised learning with error correction (corrective learning).

4.1.2 Vector notation

In the following sections we deal with learning methods for perceptrons. To simplify the notation we adopt the following conventions. The input $(x_1, x_2, \ldots, x_n)$ to the perceptron is called the input vector. If the weights of the perceptron are the real numbers $w_1, w_2, \ldots, w_n$ and the threshold is θ, we call $\mathbf{w} = (w_1, w_2, \ldots, w_n, w_{n+1})$ with $w_{n+1} = -\theta$ the *extended weight vector* of the perceptron and $(x_1, x_2, \ldots, x_n, 1)$ the *extended input vector*. The threshold computation of a perceptron will be expressed using scalar products. The arithmetic test computed by the perceptron is thus

$$\mathbf{w} \cdot \mathbf{x} \geq \theta\,,$$

if $\mathbf{w}$ and $\mathbf{x}$ are the weight and input vectors, and

$$\mathbf{w} \cdot \mathbf{x} \geq 0$$

if $\mathbf{w}$ and $\mathbf{x}$ are the extended weight and input vectors. It will always be clear from the context whether normal or extended vectors are being used.

If, for example, we are looking for the weights and threshold needed to implement the AND function with a perceptron, the input vectors and their associated outputs are

$$\begin{aligned}(0,0) &\mapsto 0,\\ (0,1) &\mapsto 0,\\ (1,0) &\mapsto 0,\\ (1,1) &\mapsto 1.\end{aligned}$$

If a perceptron with threshold zero is used, the input vectors must be extended and the desired mappings are

$$\begin{aligned}(0,0,1) &\mapsto 0,\\ (0,1,1) &\mapsto 0,\\ (1,0,1) &\mapsto 0,\\ (1,1,1) &\mapsto 1.\end{aligned}$$

A perceptron with three still unknown weights (w_1, w_2, w_3) can carry out this task.

4.1.3 Absolute linear separability

The proof of convergence of the perceptron learning algorithm assumes that each perceptron performs the test $\mathbf{w} \cdot \mathbf{x} > 0$. So far we have been working with perceptrons which perform the test $\mathbf{w} \cdot \mathbf{x} \geq 0$. We must just show that both classes of computing units are equivalent when the training set is finite, as is always the case in learning problems. We need the following definition.

Definition 3. *Two sets A and B of points in an n-dimensional space are called absolutely linearly separable if $n+1$ real numbers $w_1, \ldots, w_{n+1}$ exist such that every point $(x_1, x_2, \ldots, x_n) \in A$ satisfies $\sum_{i=1}^n w_i x_i > w_{n+1}$ and every point $(x_1, x_2, \ldots, x_n) \in B$ satisfies $\sum_{i=1}^n w_i x_i < w_{n+1}$*

If a perceptron with threshold zero can linearly separate two finite sets of input vectors, then only a small adjustment to its weights is needed to obtain an absolute linear separation. This is a direct corollary of the following proposition.

Proposition 7. *Two finite sets of points, A and B, in n-dimensional space which are linearly separable are also absolutely linearly separable.*

Proof. Since the two sets are linearly separable, weights $w_1, \ldots, w_{n+1}$ exist for which, without loss of generality, the inequality

$$\sum_{i=1}^{n} w_i a_i \geq w_{n+1}$$

holds for all points $(a_1, \ldots, a_n) \in A$ and

$$\sum_{i=1}^{n} w_i b_i < w_{n+1}$$

for all points $(b_1, \ldots, b_n) \in B$. Let $\varepsilon = \max\{\sum_{i=1}^n w_i b_i - w_{n+1} | (b_1, \ldots, b_n) \in B\}$. It is clear that $\varepsilon < \varepsilon/2 < 0$. Let $w' = w_{n+1} + \varepsilon/2$. For all points in A it holds that

$$\sum_{i=1}^{n} w_i a_i - (w' - \frac{1}{2}\varepsilon) \geq 0.$$

This means that

$$\sum_{i=1}^{n} w_i a_i - w' \geq -\frac{1}{2}\varepsilon > 0 \Rightarrow \sum_{i=1}^{n} w_i a_i > w'. \tag{4.1}$$

For all points in B a similar argument holds since

$$\sum_{i=1}^{n} w_i b_i - w_{n+1} = \sum_{i=1}^{n} w_i b_i - (w' - \frac{1}{2}\varepsilon) \leq \varepsilon,$$

and from this we deduce

$$\sum_{i=1}^{n} w_i b_i - w' \leq \frac{1}{2}\varepsilon < 0. \tag{4.2}$$

Equations (4.1) and (4.2) show that the sets A and B are absolutely linearly separable. If two sets are linearly separable in the absolute sense, then they are, of course, linearly separable in the conventional sense. □

4.1.4 The error surface and the search method

A usual approach for starting the learning algorithm is to initialize the network weights randomly and to improve these initial parameters, looking at each step to see whether a better separation of the training set can be achieved. In this section we identify points $(x_1, x_2, \ldots, x_n)$ in n-dimensional space with the vector $\mathbf{x}$ with the same coordinates.

Definition 4. *The open (closed) positive half-space associated with the n-dimensional weight vector $\mathbf{w}$ is the set of all points $\mathbf{x} \in \mathbb{R}^n$ for which $\mathbf{w} \cdot \mathbf{x} > 0$ $(\mathbf{w} \cdot \mathbf{x} \geq 0)$. The open (closed) negative half-space associated with $\mathbf{w}$ is the set of all points $\mathbf{x} \in \mathbb{R}^n$ for which $\mathbf{w} \cdot \mathbf{x} < 0$ $(\mathbf{w} \cdot \mathbf{x} \leq \mathbf{0})$.*

We omit the adjectives "closed" or "open" whenever it is clear from the context which kind of linear separation is being used.

Let P and N stand for two finite sets of points in $\mathbb{R}^n$ which we want to separate linearly. A weight vector is sought so that the points in P belong to its associated positive half-space and the points in N to the negative half-space. The *error* of a perceptron with weight vector $\mathbf{w}$ is the number of incorrectly classified points. The learning algorithm must minimize this error function $E(\mathbf{w})$. One possible strategy is to use a local greedy algorithm which works by computing the error of the perceptron for a given weight vector, looking then for a direction in weight space in which to move, and updating the weight vector by selecting new weights in the selected search direction. We can visualize this strategy by looking at its effect in weight space.

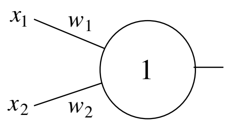


Fig. 4.4. Perceptron with constant threshold

Let us take as an example a perceptron with constant threshold $\theta = 1$ (Figure 4.4). We are looking for two weights, w_1 and w_2, which transform the perceptron into a binary AND gate. We can show graphically the error function for all combinations of the two variable weights. This has been done in Figure 4.5 for values of the weights between -0.5 and 1.5. The solution region is the triangular area in the middle. The learning algorithm should reach this region starting from any other point in weight space. In this case, it is possible to descend from one surface to the next using purely local decisions at each point.

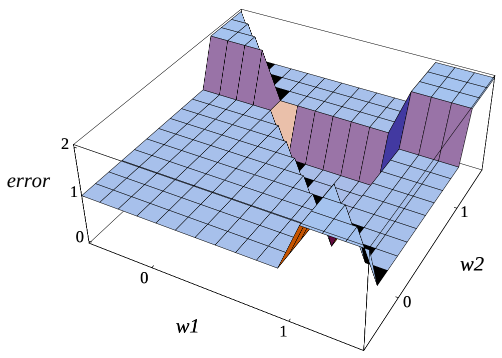


Fig. 4.5. Error function for the AND function

Figure 4.6 shows the different regions of the error function as seen from "above". The solution region is a triangle with error level 0. For the other regions the diagram shows their corresponding error count. The figure illustrates an iterative search process that starts at $\mathbf{w}_0$, goes through $\mathbf{w}_1$, $\mathbf{w}_2$, and finally reaches the solution region at $\mathbf{w}^*$. Later on, this visualization will help us to understand the computational complexity of the perceptron learning algorithm.

The optimization problem we are trying to solve can be understood as descent on the error surface but also as a search for an inner point of the solution region. Let $N = \{(0,0),(1,0),(0,1)\}$ and $P = \{(1,1)\}$ be two sets of points to be separated absolutely. The set P must be classified in the positive

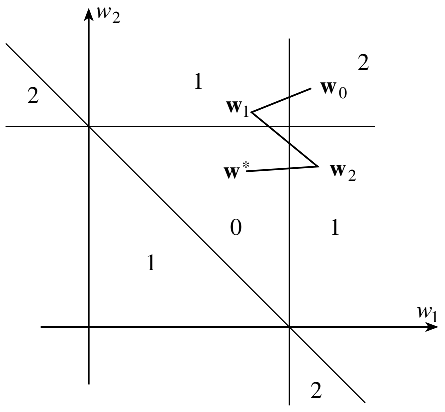


Fig. 4.6. Iteration steps to the region of minimal error

and the set N in the negative half-space. This is the separation corresponding to the AND function.

Three weights w_1, w_2, and $w_3 = -\theta$ are needed to implement the desired separation with a generic perceptron. The first step is to extend the input vectors with a third coordinate $x_3 = 1$ and to write down the four inequalities that must be fulfilled:

$$(0,0,1) \cdot (w_1, w_2, w_3) < 0 \tag{4.3}$$

$$(1,0,1) \cdot (w_1, w_2, w_3) < 0 \tag{4.4}$$

$$(0,1,1) \cdot (w_1, w_2, w_3) < 0 \tag{4.5}$$

$$(1,1,1) \cdot (w_1, w_2, w_3) > 0 \tag{4.6}$$

These equations can be written in the following simpler matrix form:

$$\begin{pmatrix} 0 & 0 & -1 \\ -1 & 0 & -1 \\ 0 & -1 & -1 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} > \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}. \tag{4.7}$$

This can be written as

$$\mathbf{Aw} > \mathbf{0}.$$

where $\mathbf{A}$ is the 4×3 matrix of Equation (4.7) and $\mathbf{w}$ the weight vector (written as a column vector). The learning problem is to find the appropriate weight vector $\mathbf{w}$.

Equation (4.7) describes all points in the interior of a convex polytope. The sides of the polytope are delimited by the planes defined by each of the inequalities (4.3)–(4.6). Any point in the interior of the polytope represents a solution for the learning problem.

We saw that the solution region for the AND function has a triangular shape when the threshold is fixed at 1. In Figure 4.7 we have a three-dimensional view of the whole solution region when the threshold (i.e., w_3) is allowed to change. The solution region of Figure 4.5 is just a cut of the solution polytope of Figure 4.7 at $w_3 = -1$. The shaded surface represents the present cut, which is similar to any other cut we could make to the polytope for different values of the threshold.

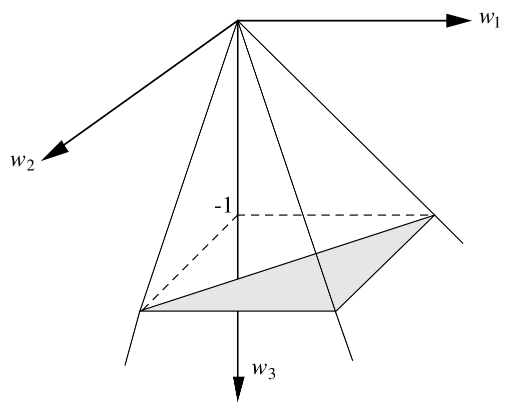


Fig. 4.7. Solution polytope for the AND function in weight space

We can see that the polytope is unbounded in the direction of negative w_3. This means that the absolute value of the threshold can become as large as we want and we will still find appropriate combinations of w_1 and w_2 to compute the AND function. The fact that we have to look for interior points of polytopes for the solution of the learning problem, is an indication that linear programming methods could be used. We will elaborate on this idea later on.

4.2 Algorithmic learning

We are now in a position to introduce the perceptron learning algorithm. The training set consists of two sets, P and N, in n-dimensional extended input space. We look for a vector $\mathbf{w}$ capable of absolutely separating both sets, so that all vectors in P belong to the open positive half-space and all vectors in N to the open negative half-space of the linear separation.

Algorithm 4.2.1 *Perceptron learning*

```
start:    The weight vector $\mathbf{w}_0$ is generated randomly,
          set $t := 0$
test:     A vector $\mathbf{x} \in P \cup N$ is selected randomly,
          if $\mathbf{x} \in P$ and $\mathbf{w}_t \cdot \mathbf{x} > 0$ go to test,
          if $\mathbf{x} \in P$ and $\mathbf{w}_t \cdot \mathbf{x} \leq 0$ go to add,
          if $\mathbf{x} \in N$ and $\mathbf{w}_t \cdot \mathbf{x} < 0$ go to test,
          if $\mathbf{x} \in N$ and $\mathbf{w}_t \cdot \mathbf{x} \geq 0$ go to subtract.
add:      set $\mathbf{w}_{t+1} = \mathbf{w}_t + \mathbf{x}$ and $t := t + 1$, goto test
subtract: set $\mathbf{w}_{t+1} = \mathbf{w}_t - \mathbf{x}$ and $t := t + 1$, goto test
```

This algorithm [312] makes a correction to the weight vector whenever one of the selected vectors in P or N has not been classified correctly. The perceptron convergence theorem guarantees that if the two sets P and N are linearly separable the vector $\mathbf{w}$ is updated only a finite number of times. The routine can be stopped when all vectors are classified correctly. The corresponding test must be introduced in the above pseudocode to make it stop and to transform it into a fully-fledged algorithm.

4.2.1 Geometric visualization

There are two alternative ways to visualize perceptron learning, one more effective than the other. Given the two sets of points $P \in \mathbb{R}^2$ and $N \in \mathbb{R}^2$ to be separated, we can visualize the linear separation in input space, as in Figure 4.8, or in extended input space. In the latter case we extend the input vectors and look for a linear separation through the origin, that is, a plane with equation $w_1x_1 + w_2x_2 + w_3 = 0$. The vector normal to this plane is the weight vector $\mathbf{w} = (w_1, w_2, w_3)$. Figure 4.9 illustrates this approach.

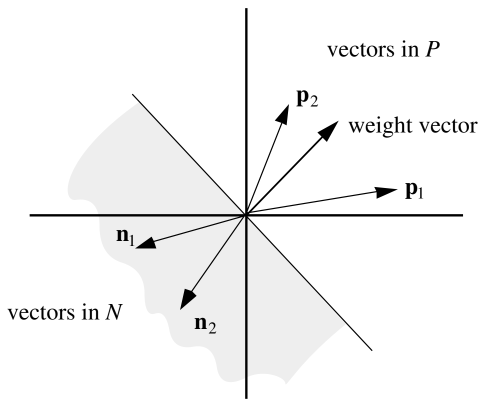


Fig. 4.8. Visualization in input space

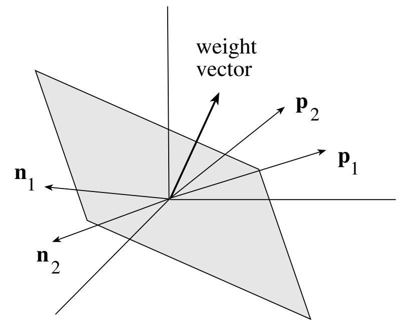


Fig. 4.9. Visualization in extended input space

We are thus looking for a weight vector $\mathbf{w}$ with a positive scalar product with all the extended vectors represented by the points in P and with a negative product with the extended vectors represented by the points in N. Actually, we will deal with this problem in a more general way. Assume that P and N are sets of n-dimensional vectors and a weight vector $\mathbf{w}$ must be found, such that $\mathbf{w} \cdot \mathbf{x} > 0$ holds for all $x \in P$ and $\mathbf{w} \cdot \mathbf{x} < 0$ holds for all $x \in N$.

The perceptron learning algorithm starts with a randomly chosen vector $\mathbf{w}_0$. If a vector $\mathbf{x} \in P$ is found such that $\mathbf{w} \cdot \mathbf{x} < 0$, this means that the angle between the two vectors is greater than 90 degrees. The weight vector must be rotated in the direction of $\mathbf{x}$ to bring this vector into the positive half-space defined by $\mathbf{w}$. This can be done by adding $\mathbf{w}$ and $\mathbf{x}$, as the perceptron learning algorithm does. If $\mathbf{x} \in N$ and $\mathbf{w} \cdot \mathbf{x} > 0$, then the angle between $\mathbf{x}$ and $\mathbf{w}$ is less than 90 degrees. The weight vector must be rotated away from $\mathbf{x}$. This is done by subtracting $\mathbf{x}$ from $\mathbf{w}$. The vectors in P rotate the weight vector in one direction, the vectors in N rotate the negative weight vector in another. If a solution exists it can be found after a finite number of steps. A good initial heuristic is to start with the average of the positive input vectors minus the average of the negative input vectors. In many cases this yields an initial vector near the solution region.

In perceptron learning we are not necessarily dealing with normalized vectors, so that every update of the weight vector of the form $\mathbf{w} \pm \mathbf{x}$ rotates the weight vector by a different angle. If $\mathbf{x} \in P$ and $\|\mathbf{x}\| \gg \|\mathbf{w}\|$ the new weight vector $\mathbf{w} + \mathbf{x}$ is almost equal to $\mathbf{x}$. This effect and the way perceptron learning works can be seen in Figure 4.10. The initial weight vector is updated by adding $\mathbf{x}_1$, $\mathbf{x}_3$, and $\mathbf{x}_1$ again to it. After each correction the weight vector is rotated in one or the other direction. It can be seen that the vector $\mathbf{w}$ becomes larger after each correction in this example. Each correction rotates the weight vector by a smaller angle until the correct linear separation has been found. After the initial updates, successive corrections become smaller and the algorithm "fine tunes" the position of the weight vector. The learning rate, the rate of change of the vector $\mathbf{w}$, becomes smaller in time and if we

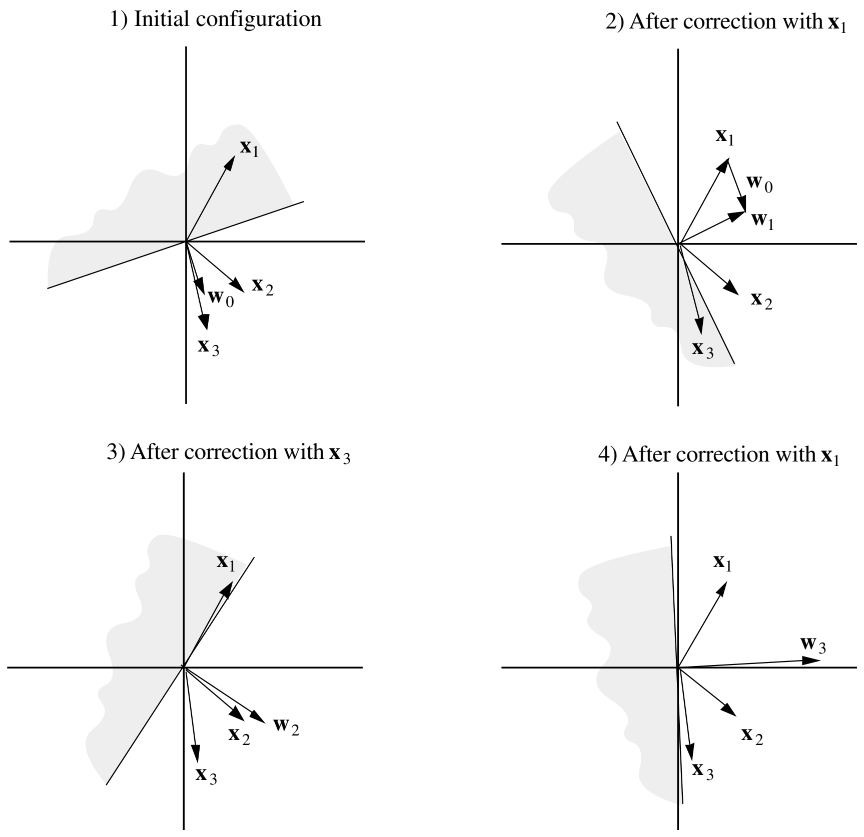


Fig. 4.10. Convergence behavior of the learning algorithm

keep on training, even after the vectors have already been correctly separated, it approaches zero. Intuitively we can think that the learned vectors are increasing the "inertia" of the weight vector. Vectors lying just outside of the positive region are brought into it by rotating the weight vector just enough to correct the error.

This is a typical feature of many learning algorithms for neural networks. They make use of a so-called *learning constant*, which is brought to zero during the learning process to consolidate what has been learned. The perceptron learning algorithm provides a kind of automatic learning constant which determines the degree of adaptivity (the so-called *plasticity* of the network) of the weights.

4.2.2 Convergence of the algorithm

The convergence proof of the perceptron learning algorithm is easier to follow by keeping in mind the visualization discussed in the previous section.

Proposition 8. *If the sets P and N are finite and linearly separable, the perceptron learning algorithm updates the weight vector $\mathbf{w}_t$ a finite number of times. In other words: if the vectors in P and N are tested cyclically one after the other, a weight vector $\mathbf{w}_t$ is found after a finite number of steps t which can separate the two sets.*

Proof. We can make three simplifications without losing generality:

i) The sets P and N can be joined in a single set $P' = P \cup N^-$, where the set N^- consists of the negated elements of N.
ii) The vectors in P' can be normalized, because if a weight vector $\mathbf{w}$ is found so that $\mathbf{w} \cdot \mathbf{x} > 0$ this is also valid for any other vector $\eta\mathbf{x}$, where η is a constant.
iii) The weight vector can also be normalized. Since we assume that a solution for the linear separation problem exists, we call $\mathbf{w}^*$ a normalized solution vector.

Assume that after $t+1$ steps the weight vector $\mathbf{w}_{t+1}$ has been computed. This means that at time t a vector $\mathbf{p}_i$ was incorrectly classified by the weight vector $\mathbf{w}_t$ and so $\mathbf{w}_{t+1} = \mathbf{w}_t + \mathbf{p}_i$.

The cosine of the angle ρ between $\mathbf{w}_{t+1}$ and $\mathbf{w}^*$ is

$$\cos\rho = \frac{\mathbf{w}^* \cdot \mathbf{w}_{t+1}}{\|\mathbf{w}_{t+1}\|} \tag{4.8}$$

For the expression in the numerator we know that

$$\begin{aligned}\mathbf{w}^*\mathbf{w}_{t+1} &= \mathbf{w}^* \cdot (\mathbf{w}_t + \mathbf{p}_i)\\ &= \mathbf{w}^* \cdot \mathbf{w}_t + \mathbf{w}^* \cdot \mathbf{p}_i\\ &\geq \mathbf{w}^* \cdot \mathbf{w}_t + \delta\end{aligned}$$

with $\delta = \min\{\mathbf{w}^* \cdot \mathbf{p} \mid \forall \mathbf{p} \in P'\}$. Since the weight vector $\mathbf{w}^*$ defines an absolute linear separation of P and N we know that $\delta > 0$. By induction we obtain

$$\mathbf{w}^* \cdot \mathbf{w}_{t+1} \geq \mathbf{w}^* \cdot \mathbf{w}_0 + (t+1)\delta. \tag{4.9}$$

On the other hand for the term in the denominator of (4.8) we know that

$$\begin{aligned}\|\mathbf{w}_{t+1}\|^2 &= (\mathbf{w}_t + \mathbf{p}_i) \cdot (\mathbf{w}_t + \mathbf{p}_i)\\ &= \|\mathbf{w}_t\|^2 + 2\mathbf{w}_t \cdot \mathbf{p}_i + \|\mathbf{p}_i\|^2\end{aligned}$$

Since $\mathbf{w}_t \cdot \mathbf{p}_i$ is negative or zero (otherwise we would have not corrected $\mathbf{w}_t$ using $\mathbf{p}_i$) we can deduce that

$$\begin{aligned}\|\mathbf{w}_{t+1}\|^2 &\leq \|\mathbf{w}_t\|^2 + \|\mathbf{p}_i\|^2\\ &\leq \|\mathbf{w}_t\|^2 + 1\end{aligned}$$

because all vectors in P have been normalized. Induction then gives us

$$\|\mathbf{w}_{t+1}\|^2 \leq \|\mathbf{w}_0\|^2 + (t+1). \qquad (4.10)$$

From (4.9) and (4.10) and Equation (4.8) we get the inequality

$$\cos\rho \geq \frac{\mathbf{w}^* \cdot \mathbf{w}_0 + (t+1)\delta}{\sqrt{\|\mathbf{w}_0\|^2 + (t+1)}}$$

The right term grows proportionally to $\sqrt{t}$ and, since δ is positive, it can become arbitrarily large. However, since $\cos\rho \leq 1$, t must be bounded by a maximum value. Therefore, the number of corrections to the weight vector must be finite. □

The proof shows that perceptron learning works by bringing the initial vector $\mathbf{w}_0$ sufficiently close to $\mathbf{w}^*$ (since $\cos\rho$ becomes larger and ρ proportionately smaller).

4.2.3 Accelerating convergence

Although the perceptron learning algorithm converges to a solution, the number of iterations can be very large if the input vectors are not normalized and are arranged in an unfavorable way.

There are faster methods to find the weight vector appropriate for a given problem. When the perceptron learning algorithm makes a correction, an input vector $\mathbf{x}$ is added or subtracted from the weight vector $\mathbf{w}$. The search direction is given by the vector $\mathbf{x}$. Each input vector corresponds to the border of one region of the error function defined on weight space. The direction of $\mathbf{x}$ is orthogonal to the step defined by $\mathbf{x}$ on the error surface. The weight vector is displaced in the direction of $\mathbf{x}$ until it "falls" into a region with smaller error.

We can illustrate the dynamics of perceptron learning using the error surface for the OR function as an example. The input $(1,1)$ must produce the output 1 (for simplicity we fix the threshold of the perceptron to 1). The two weights w_1 and w_2 must fulfill the inequality $w_1 + w_2 \geq 1$. Any other combination produces an error. The contribution to the total error is shown in Figure 4.11 as a step in the error surface. If the initial weight vector lies in the triangular region with error 1, it must be brought up to the verge of the region with error 0. This can be done by adding the vector (1,1) to $\mathbf{w}$. However, if the input vector is, for example, $(0.1, 0.1)$, it should be added a few times before the weight combination (w_1, w_2) falls to the region of error 0. In this case we would like to make the correction in a single iteration.

These considerations provide an improvement for the perceptron learning algorithm: if at iteration t the input vector $\mathbf{x} \in P$ is classified erroneously, then we have $\mathbf{w}_t \cdot \mathbf{x} \leq 0$. The error δ can be defined as

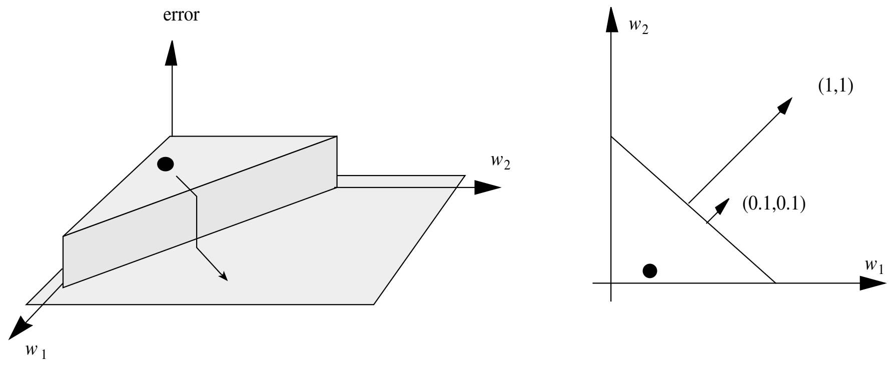


Fig. 4.11. A step on the error surface

$$\delta = -\mathbf{w}_t \cdot \mathbf{x}.$$

The new weight vector $\mathbf{w}_{t+1}$ is calculated as follows:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \frac{\delta + \varepsilon}{\|\mathbf{x}\|^2}\mathbf{x},$$

where ε denotes a small positive real number. The classification of $\mathbf{x}$ has been corrected in one step because

$$\begin{aligned} \mathbf{w}_{t+1} \cdot \mathbf{x} &= (\mathbf{w}_t + \frac{\delta + \varepsilon}{\|\mathbf{x}\|^2}\mathbf{x}) \cdot \mathbf{x} \\ &= \mathbf{w}_t \cdot \mathbf{x} + (\delta + \varepsilon) \\ &= -\delta + \delta + \varepsilon \\ &= \varepsilon > 0 \end{aligned}$$

The number ε guarantees that the new weight vector just barely skips over the border of the region with a higher error. The constant ε should be made small enough to avoid skipping to another region whose error is higher than the current one. When $\mathbf{x} \in N$ the correction step is made similarly, but using the factor $\delta - \varepsilon$ instead of $\delta + \varepsilon$.

The accelerated algorithm is an example of *corrective learning*: We do not just "reinforce" the weight vector, but completely correct the error that has been made. A variant of this rule is correction of the weight vector using a proportionality constant γ as the learning factor, in such a way that at each update the vector $\gamma(\delta + \varepsilon)\mathbf{x}$ is added to $\mathbf{w}$. The learning constant falls to zero as learning progresses.

4.2.4 The pocket algorithm

If the learning set is not linearly separable the perceptron learning algorithm does not terminate. However, in many cases in which there is no perfect linear separation, we would like to compute the linear separation which correctly classifies the largest number of vectors in the positive set P and the negative set N. Gallant proposed a very simple variant of the perceptron learning algorithm capable of computing a good approximation to this ideal linear separation. The main idea of the algorithm is to store the best weight vector found so far by perceptron learning (in a "pocket") while continuing to update the weight vector itself. If a better weight vector is found, it supersedes the one currently stored and the algorithm continues to run [152].

Algorithm 4.2.2 *Pocket algorithm*

start: Initialize the weight vector $\mathbf{w}$ randomly. Define a "stored" weight vector $\mathbf{w}_s = \mathbf{w}$. Set h_s, the history of $\mathbf{w}_s$, to zero.

iterate: Update $\mathbf{w}$ using a single iteration of the perceptron learning algorithm. Keep track of the number h of consecutively successfully tested vectors. If at any moment $h > h_s$, substitute $\mathbf{w}_s$ with $\mathbf{w}$ and h_s with h. Continue iterating.

The algorithm can occasionally change a good stored weight vector for an inferior one, since only information from the last run of selected examples is considered. The probability of this happening, however, becomes smaller and smaller as the number of iterations grows. If the training set is finite and the weights and vectors are rational, it can be shown that this algorithm converges to an optimal solution with probability 1 [152].

4.2.5 Complexity of perceptron learning

The perceptron learning algorithm selects a search direction in weight space according to the incorrect classification of the last tested vector and does not make use of global information about the shape of the error function. It is a greedy, local algorithm. This can lead to an exponential number of updates of the weight vector.

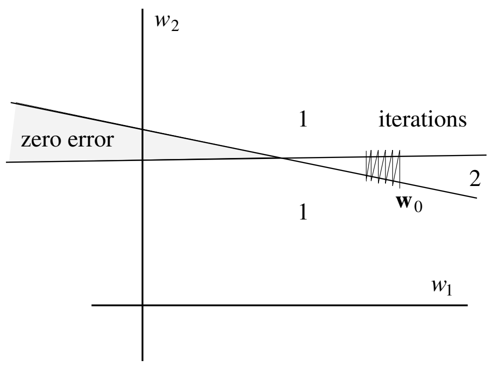


Fig. 4.12. Worst case for perceptron learning (weight space)

Figure 4.12 shows the different error regions in a worst case scenario. The region with error 0 is bounded by two lines which meet at a small angle. Starting the learning algorithm at point $\mathbf{w}_0$, the weight updates will lead to a search path similar to the one shown in the figure. In each new iteration a new weight vector is computed, in such a way that one of two vectors is classified correctly. However, each of these corrections leads to the other vector being incorrectly classified. The iteration jumps from one region with error 1 to the other one. The algorithm converges only after a certain number of iterations, which can be made arbitrarily large by adjusting the angle at which the lines meet.

Figure 4.12 corresponds to the case in which two almost antiparallel vectors are to be classified in the same half-space (Figure 4.13). An algorithm which rotates the separation line in one of the two directions (like perceptron learning) will require more and more time when the angle between the two vectors approaches 180 degrees.

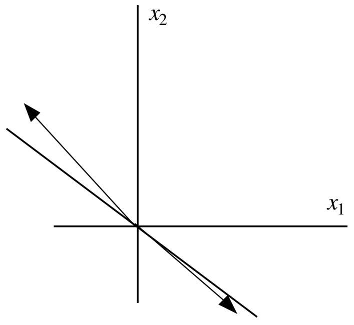


Fig. 4.13. Worst case for perceptron learning (input space)

This example is a good illustration of the advantages of visualizing learning algorithms in both the input space and its dual, weight space. Figure 4.13 shows the concrete problem and Figure 4.12 illustrates why it is difficult to solve.

4.3 Linear programming

A set of input vectors to be separated by a perceptron in a positive and a negative set defines a convex polytope in weight space, whose inner points represent all admissible weight combinations for the perceptron. The perceptron learning algorithm finds a solution when the interior of the polytope is not void. Stated differently: if we want to train perceptrons to classify patterns, we must solve an inner point problem. Linear programming can deal with this kind of task.

4.3.1 Inner points of polytopes

Linear programming was developed to solve the following generic problem: Given a set of n variables $x_1, x_2, \ldots, x_n$ a function $c_1x_1+c_2x_2+\cdots+c_nx_n$ must be maximized (or minimized). The variables must obey certain constraints given by linear inequalities of the form

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &\leq b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &\leq b_2 \\
&\vdots\ \vdots\ \vdots \\
a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &\leq b_m
\end{aligned}$$

All m linear constraints can be summarized in the matrix inequality $\mathbf{Ax} \leq \mathbf{b}$, in which $\mathbf{x}$ and $\mathbf{b}$ respectively represent n-dimensional and m-dimensional column vectors and $\mathbf{A}$ is a $m \times n$ matrix. It is also necessary that $\mathbf{x} \geq \mathbf{0}$, which can always be guaranteed by introducing additional variables.

As in the case of a perceptron, the m inequalities define a convex polytope of feasible values for the variables $x_1, x_2, \ldots, x_n$. If the optimization problem has a solution, this is found at one of the vertices of the polytope. Figure 4.14 shows a two-dimensional example. The shaded polygon is the feasible region. The function to be optimized is represented by the line normal to the vector c. Finding the point where this linear function reaches its maximum corresponds to moving the line, without tilting it, up to the farthest position at which it is still in contact with the feasible region, in our case ξ. It is intuitively clear that when one or more solutions exist, one of the vertices of the polytope is one of them.

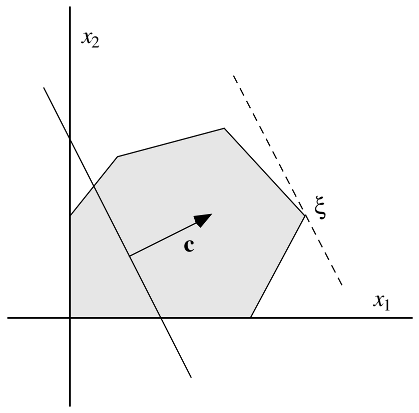


Fig. 4.14. Feasible region for a linear optimization problem

The well-known simplex algorithm of linear programming starts at a vertex of the feasible region and jumps to another neighboring vertex, always moving in a direction in which the function to be optimized increases. In the worst case an exponential number of vertices in the number of inequalities m has to be traversed before a solution is found. On average, however, the simplex algorithm is not so inefficient. In the case of Figure 4.14 the optimal solution can be found in two steps by starting at the origin and moving to the right. To determine the next vertex to be visited, the simplex algorithm uses as a criterion the length of the projection of the gradient of the function to be optimized on the edges of the polytope. It is in this sense a gradient algorithm. The algorithm can be inefficient because the search for the optimum is constrained to be carried out moving only along the edges of the polytope. If the number of delimiting surfaces is large, a better alternative is to go right through the middle of the polytope.

4.3.2 Linear separability as linear optimization

The simplex algorithm and its variants need to start at a point in the feasible region. In many cases it can be arranged to start at the origin. If the feasible region does not contain the origin as one of its vertices, a feasible point must be found first. This problem can be transformed into a linear program.

Let $\mathbf{A}$ represent the $m \times n$ matrix of coefficients of the linear constraints and $\mathbf{b}$ an m-dimensional column vector. Assume that we are looking for an n-dimensional column vector $\mathbf{x}$ such that $\mathbf{Ax} \leq \mathbf{b}$. This condition is fulfilled only by points in the feasible region. To simplify the problem, assume that $\mathbf{b} \geq \mathbf{0}$ and that we are looking for vectors $\mathbf{x} \geq \mathbf{0}$. Introducing the column vector $\mathbf{y}$ of m additional slack variables $(y_1, \ldots, y_m)$, the inequality $\mathbf{Ax} \leq \mathbf{b}$ can be transformed into the equality $\mathbf{Ax}+\mathbf{Iy} = \mathbf{b}$, where $\mathbf{I}$ denotes the $m \times m$ identity matrix. The linear program to be solved is then

$$\min\{\sum_{i=1}^{m} y_i | \mathbf{Ax} + \mathbf{Iy} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}, \mathbf{y} \geq \mathbf{0}\}.$$

An initial feasible solution for the problem is $\mathbf{x} = \mathbf{0}$ and $\mathbf{y} = \mathbf{b}$. Starting from here an iterative algorithm looks for the minimum of the sum of the slack variables. If the minimum is negative the original problem does not have a solution and the feasible region of $\mathbf{Ax} \leq \mathbf{b}$ is void. If the minimum is zero, the value of $\mathbf{x}$ determined during the optimization is an inner point of the feasible region (more exactly, a point at its boundary).

The conditions $\mathbf{x} \geq \mathbf{0}$ and $\mathbf{b} \geq 0$ can be omitted and additional transformations help to transform the more general problem to the canonical form discussed here [157].

Inner points of convex polytopes, defined by separating hyperplanes, can thus be found using linear programming algorithms. Since the computation of the weight vector for a perceptron corresponds to the computation of inner points of convex polytopes, this means that perceptron learning can also be handled in this way. If two sets of vectors are not linearly separable, the linear programming algorithm can detect it. The complexity of linearly separating points in an input space is thus bounded by the complexity of solving linear programming problems.

The perceptron learning algorithm is not the most efficient method for perceptron learning, since the number of steps can grow exponentially in the worst case. In the case of linear programming, theoreticians have succeeded in crafting algorithms which need a polynomial number of iterations and return the optimal solution or an indication that it does not exist.

4.3.3 Karmarkar's algorithm

In 1984 a fast polynomial time algorithm for linear programming was proposed by Karmarkar [236]. His algorithm starts at an inner point of the solution region and proceeds in the direction of steepest ascent (if maximizing), taking care not to step out of the feasible region.

Figure 4.15 schematically shows how the algorithm works. The algorithm starts with a canonical form of the linear programming problem in which the additional constraint $x_1 + x_2 + \cdots + x_n = 1$ is added to the basic constraints $\mathbf{Ax} \geq \mathbf{0}$, where $x_1, \ldots, x_n$ are the variables in the problem. Some simple transformations can bring the original problem into this form. The point $\mathbf{e} = \frac{1}{n}(1, 1, \ldots, 1)$ is considered the middle of the solution polytope and each iteration step tries to transform the original problem in such a way that this is always the starting point.

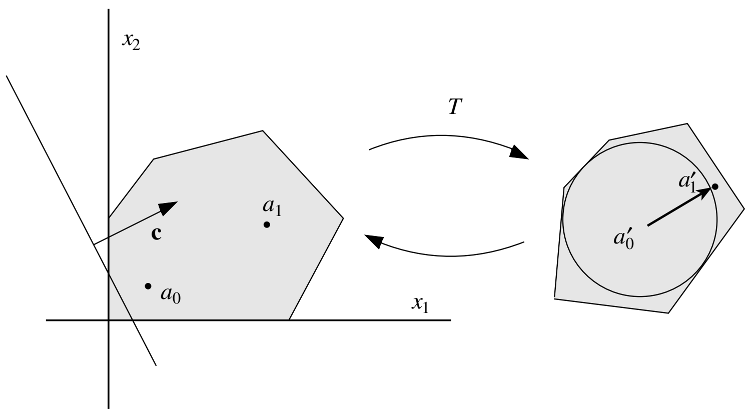


Fig. 4.15. Transformation of the solution polytope

An initial point a_0 is selected in the interior of the solution polytope and then brought into the middle $\mathbf{e}$ of the transformed feasible region using a projective transformation T. A projective transformation maps each point $\mathbf{x}$ in the hyperplane $x_1 + x_2 + \cdots + x_n = 1$ to a point $\mathbf{x}'$ in another hyperplane, whereby the line joining $\mathbf{x}$ and $\mathbf{x}'$ goes through a predetermined point p. The transformation is applied on the initial point a_0, the matrix $\mathbf{A}$ of linear constraints and also to the linear function $\mathbf{c}^T\mathbf{x}$ to be optimized. After the transformation, the radius of the largest sphere with center a'_0 and inside the transformed feasible region is computed. Starting at the center of the sphere a new point in the direction of the transformed optimizing direction $\mathbf{c}'$ is computed. The step length is made shorter than the computed maximal radius by a small factor, to avoid reaching the surface of the solution polytope. The new point a'_1 computed in this way is a feasible point and is also strictly in the interior of the solution polytope. The point a'_1 is transformed back to the original space using the inverse projective transformation T^{-1} and a new iteration can start again from this point. This basic step is repeated, periodically testing whether a vertex of the polytope is close enough and optimal. At this moment the algorithm stops and takes this vertex as the solution of the problem (Figure 4.16). Additionally, a certain checking must be done in each iteration to confirm that a solution to the optimization problem exists and that the cost function is not unbounded.

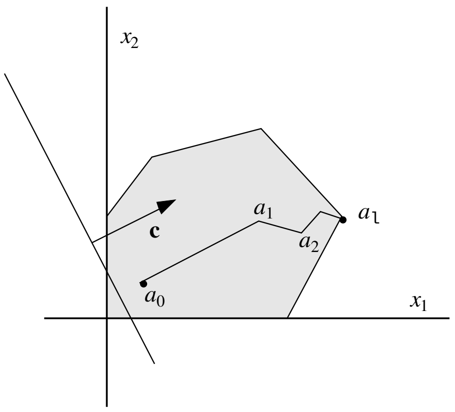


Fig. 4.16. Example of a search path for Karmarkar's algorithm

In the worst case Karmarkar's algorithm executes in a number of iterations proportional to $n^{3.5}$, where n is the number of variables in the problem and other factors are kept constant. Some published modifications of Karmarkar's algorithm are still more efficient but start beating the simplex method in the average case only when the number of variables and constraints becomes relatively large, since the computational overhead for a small number of constraints is not negligible [257].

The existence of a polynomial time algorithm for linear programming and for the solution of interior point problems shows that perceptron learning is not what is called a *hard* computational problem. Given any number of training patterns, the learning algorithm (in this case linear programming) can decide whether the problem has a solution or not. If a solution exists, it finds the appropriate weights in polynomial time at most.

4.4 Historical and bibliographical remarks

The success of the perceptron and the interest it aroused in the 1960s was a direct product of its learning capabilities, different from the hand-design approach of previous models. Later on, research in this field reached an impasse when a learning algorithm for more general networks was still unavailable.

Minsky and Papert [312] analyzed the features and limitations of the perceptron model in a rigorous way. They could show that the perceptron learning algorithm needs an exponential number of learning steps in the worst case. However, perceptron learning is in the average case fairly efficient. Mansfield showed that when the training set is selected randomly from a half-space, the number of iterations of the perceptron learning algorithm is comparable to the number of iterations needed by ellipsoid methods for linear programming (up to dimension 30) [290]. Baum had previously shown that when the learning set is picked by a non-malicious adversary, the complexity of perceptron learning is polynomial [46].

More recently the question has arisen of whether a given set of nonlinearly separable patterns can be decomposed in such a way that the largest linearly separable subset can be detected. Amaldi showed that this is an *NP*-complete problem, that is, a problem for which presumably no polynomial time algorithm exists (compare Chap. 10).

The conditions for perfect perceptron learning can be also relaxed. If the set of patterns is not linearly separable, we can look for the separation that minimizes the average quadratic error, without requiring it to be zero. In this case statistical methods or the backpropagation algorithm (Chap. 7) can be used.

After the invention of the simplex algorithm for linear programming there was a general feeling that it could be proven that one of its variants was of polynomial complexity in the number of constraints and of variables. This was due to the fact that the actual experience with the algorithm showed that in the average case a solution was found in much less than exponential time. However, in 1972 Klee and Minty [247] gave a counterexample which showed that there were situations in which the simplex method visited 2^{n-1} vertices of a feasible region with 2^n vertices. Later it was rigorously proven that the simplex method is polynomial in the average case [64]. The question of the existence of a polynomial time algorithm for linear programming was settled by Khachiyan in 1979, when he showed that a recursive construction of ellipsoids could lead to finding the optimal vertex of the feasible region in polynomial time [244]. His algorithm, however, was very computationally intensive for most of the average-sized problems and could not displace the simplex method. Karmarkar's algorithm, a further development of the ellipsoid method including some very clever transformations, aroused much interest when it was first introduced in 1984. So many variations of the original algorithm have appeared that they are collectively known as Karmarkar-type algorithms. Minimization problems with thousands of constraints can now be dealt with efficiently by these polynomial time algorithms, but since the simplex method is fast in the average case it continues to be the method of choice in medium-sized problems.

Interesting variations of perceptron learning were investigated by Fontanari and Meir, who coded the different alternatives of weight updates according to the local information available to each weight and let a population of algorithms evolve. With this kind of "evolution strategy" they found competitive algorithms similar to the standard methods [140].

Exercises

1. Implement the perceptron learning algorithm in the computer. Find the weights for an edge detection operator using this program. The input-output examples can be taken from a digitized picture of an object and another one in which only the edges of the object have been kept.
2. Give a numerical example of a training set that leads to many iterations of the perceptron learning algorithm.
3. How many vectors can we pick randomly in an n-dimensional space so that they all belong to the same half-space? Produce a numerical estimate using a computer program.
4. The perceptron learning algorithm is usually fast if the vectors to be linearly separated are chosen randomly. Choose a weight vector $\mathbf{w}$ for a perceptron randomly. Generate p points in input space and classify them in a positive or negative class according to their scalar product with $\mathbf{w}$. Now train a perceptron using this training set and measure the number of iterations needed. Make a plot of n against p for dimension up to 10 and up to 100 points.

5

Unsupervised Learning and Clustering Algorithms

5.1 Competitive learning

The perceptron learning algorithm is an example of *supervised learning.* This kind of approach does not seem very plausible from the biologist's point of view, since a teacher is needed to accept or reject the output and adjust the network weights if necessary. Some researchers have proposed alternative learning methods in which the network parameters are determined as a result of a self-organizing process. In *unsupervised learning* corrections to the network weights are not performed by an external agent, because in many cases we do not even know what solution we should expect from the network. The network itself decides what output is best for a given input and reorganizes accordingly.

We will make a distinction between two classes of unsupervised learning: *reinforcement* and *competitive learning.* In the first method each input produces a reinforcement of the network weights in such a way as to enhance the reproduction of the desired output. Hebbian learning is an example of a reinforcement rule that can be applied in this case. In competitive learning, the elements of the network compete with each other for the "right" to provide the output associated with an input vector. Only one element is allowed to answer the query and this element simultaneously inhibits all other competitors.

This chapter deals with competitive learning. We will show that we can conceive of this learning method as a generalization of the linear separation methods discussed in the previous two chapters.

5.1.1 Generalization of the perceptron problem

A single perceptron divides input space into two disjoint half-spaces. However, as we already mentioned in Chap. 3, the relative number of linearly separable Boolean functions in relation to the total number of Boolean functions converges to zero as the dimension of the input increases without bound. Therefore we would like to implement some of those not linearly separable functions using not a single perceptron but a collection of computing elements.

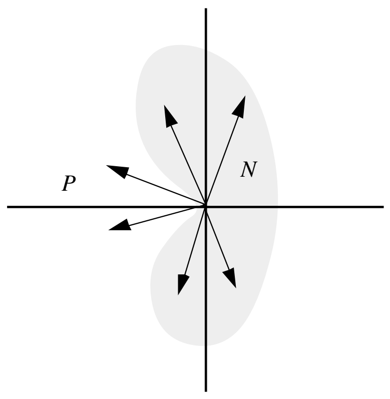


Fig. 5.1. The two sets of vectors P and N

Figure 5.1 shows a two-dimensional problem involving two sets of vectors, denoted respectively P and N. The set P consists of a more or less compact bundle of vectors. The set N consists of vectors clustered around two different regions of space.

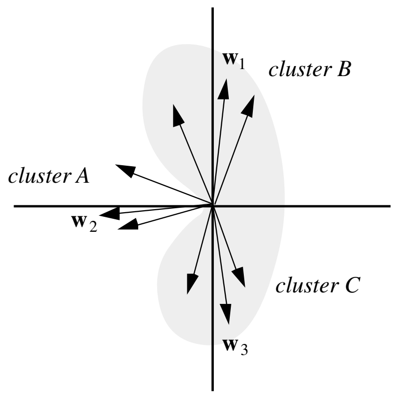


Fig. 5.2. Three weight vectors for the three previous clusters

This classification problem is too complex for a single perceptron. A weight vector $\mathbf{w}$ cannot satisfy $\mathbf{w} \cdot \mathbf{p} \geq 0$ for all vectors $\mathbf{p}$ in P and $\mathbf{w} \cdot \mathbf{n} < 0$ for all vectors $\mathbf{n}$ in N. In this situation it is possible to find three different vectors $\mathbf{w}_1$,$\mathbf{w}_2$ and $\mathbf{w}_3$ which can act as a kind of “representative” for the vectors in each of the three clusters A, B and C shown in Figure 5.2. Each one of these vectors is not very far apart from every vector in its cluster. Each weight vector corresponds to a single computing unit, which only fires when the input vector is close enough to its own weight vector.

If the number and distribution of the input clusters is known in advance, we can select a representative for each one by doing a few simple computations. However the problem is normally much more general: if the number and distribution of clusters is unknown, how can we decide how many computing units and thus how many representative weight vectors we should use? This is the well-known *clustering problem*, which arises whenever we want to classify multidimensional data sets whose deep structure is unknown. One example ot this would be the number of phonemes in speech. We can transform small segments of speech to n-dimensional data vectors by computing, for example, the energy in each of n selected frequency bands. Once this has been done, how many different patterns should we distinguish? As many as we think we perceive in English? However, there are African languages with a much richer set of articulations. It is this kind of ambiguity that must be resolved by unsupervised learning methods.

5.1.2 Unsupervised learning through competition

The solution provided in this chapter for the clustering problem is just a generalization of perceptron learning. Correction steps of the perceptron learning algorithm rotate the weight vector in the direction of the wrongly classified input vector (for vectors belonging to the positive half-space). If the problem is solvable, the weight vector of a perceptron oscillates until a solution is found.

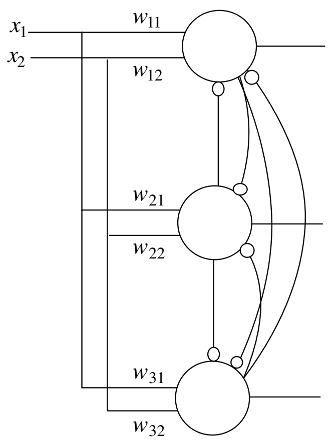


Fig. 5.3. A network of three competing units

In the case of unsupervised learning, the n-dimensional input is processed by exactly the same number of computing units as there are clusters to be individually identified. For the problem of three clusters in Figure 5.2 we could use the network shown in Figure 5.3.

The inputs x_1 and x_2 are processed by the three units in Figure 5.3. Each unit computes its weighted input, but only the unit with the largest excitation is allowed to fire a 1. The other units are inhibited by this active element through the lateral connections shown in the diagram. Deciding whether or not to activate a unit requires therefore *global information* about the state of each unit. The firing unit signals that the current input is an element of the cluster of vectors it represents. We could also think of this computation as being performed by perceptrons with variable thresholds. The thresholds are adjusted in each computation in such a way that just one unit is able to fire.

The following learning algorithm allows the identification of clusters of input vectors. We can restrict the network to units with threshold zero without losing any generality.

Algorithm 5.1.1 *Competitive learning*

Let $X = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_\ell)$ be a set of normalized input vectors in n-dimensional space which we want to classify in k different clusters. The network consists of k units, each with n inputs and threshold zero.

```
start:  The normalized weight vectors $\mathbf{w}_1, \ldots, \mathbf{w}_k$ are generated randomly.
test:   Select a vector $\mathbf{x}_j \in X$ randomly.
        Compute $\mathbf{x}_j \cdot \mathbf{w}_i$ for $i = 1, \ldots, k$.
        Select $\mathbf{w}_m$ such that $\mathbf{w}_m \cdot \mathbf{x}_j \geq \mathbf{w}_i \cdot \mathbf{x}_j$ for $i = 1, \ldots, k$.
        Continue with update.
update: Substitute $\mathbf{w}_m$ with $\mathbf{w}_m + \mathbf{x}_j$ and normalize.
        Continue with test.
```

The algorithm can be stopped after a predetermined number of steps. The weight vectors of the k units are "attracted" in the direction of the clusters in input space. By using normalized vectors we prevent one weight vector from becoming so large that it would win the competition too often. The consequence could be that other weight vectors are never updated so that they just lie unused. In the literature the units associated with such vectors are called *dead units*. The difference between this algorithm and perceptron learning is that the input set cannot be classified *a priori* in a positive or a negative set or in any of several different clusters.

Since input and weight vectors are normalized, the scalar product $\mathbf{w}_i \cdot \mathbf{x}_j$ of a weight and an input vector is equal to the cosine of the angle spanned by both vectors. The selection rule (maximum scalar product) guarantees that the weight vector $\mathbf{w}_m$ of the cluster that is updated is the one that lies closest to the tested input vector. The update rule rotates the weight vector $\mathbf{w}_m$ in the direction of $\mathbf{x}_j$. This can be done using different learning rules:

- *Update with learning constant* – The weight update is given by

$$\Delta \mathbf{w}_m = \eta \mathbf{x}_j.$$

  The learning constant η is a real number between 0 and 1. It decays to 0 as learning progresses. The *plasticity* of the network can be controlled in such a way that the corrections are more drastic in the first iterations and afterwards become more gradual.
- *Difference update* – The weight update is given by

$$\Delta \mathbf{w}_m = \eta(\mathbf{x}_j - \mathbf{w}_m).$$

  The correction is proportional to the difference of both vectors.
- *Batch update* – Weight corrections are computed and accumulated. After a number of iterations the weight corrections are added to the weights. The use of this rule guarantees some stability in the learning process.

The learning algorithm 5.1.1 uses the strategy known as *winner-takes-all*, since only one of the network units is selected for a weight update. Convergence to a good solution can be accelerated by distributing the initial weight vectors according to an adequate heuristic. For example, one could initialize the k weight vectors with k different input vectors. In this way no weight vector corresponds to a dead unit. Another possibility is monitoring the number of updates for each weight vector in order to make the process as balanced as possible (see Exercise 4). This is called *learning with conscience*.

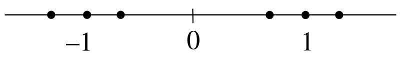
5.2 Convergence analysis

In Algorithm 5.1.1 no stop condition is included. Normally only a fixed number of iterations is performed, since it is very difficult to define the "natural" clustering for some data distributions. If there are three well-defined clusters, but only two weight vectors are used, it could well happen that the weight vectors keep skipping from cluster to cluster in a vain attempt to cover three separate regions with just two computing units. The convergence analysis of unsupervised learning is therefore much more complicated than for perceptron learning, since we are dealing with a much more general problem. Analysis of the one-dimensional case already shows the difficulties of dealing with convergence of unsupervised learning.

5.2.1 The one-dimensional case – energy function

In the one-dimensional case we deal with clusters of numbers in the real line. Let the input set be $\{-1.3, -1.0, -0.7, 0.7, 1.0, 1.3\}$.

Note that we avoid normalizing the input or the weights, since this would make no sense in the one-dimensional case. There are two well-defined clusters



centered at -1 and 1 respectively. This clustering must be identified by the network shown in Figure 5.4, which consists of two units, each with one weight. A possible solution is $\alpha = -1$ and $\beta = 1$. The winning unit inhibits the other unit.

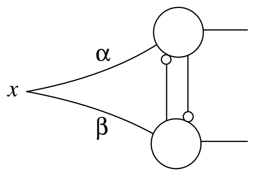


Fig. 5.4. Network for the one-dimensional case

If the learning algorithm is started with α negative and small and β positive and small, it is easy to see that the cluster of negative numbers will attract α and the cluster of positive numbers will attract β. We should expect to see α converge to -1 and β to 1. This means that one of the weights converges to the centroid of the first cluster and the other weight to the centroid of the second. This attraction can be modeled as a kind of force. Let x be a point in a cluster and α_0 the current weight of the first unit. The attraction of x on the weight is given by

$$F_x(\alpha_0) = \gamma(x - \alpha_0)\,,$$

where γ is a constant. Statisticians speak of a “potential” or “inertia” associated with this force. We will call the potential associated with F_x the *energy function* corresponding to the learning task. The energy function for our example is given by

$$E_x(\alpha_0) = \int -\gamma(x - \alpha_0)d\alpha_0 = \frac{\gamma}{2}(x - \alpha_0)^2 + C,$$

where C is an integration constant and the integration is performed over cluster 1. Note that in this case we compute the form of the function for a given α_0 under the assumption that all points of cluster 1 are nearer to α_0 than to the second weight β. The energy function is quadratic with a well-defined global minimum, since in the discrete case the integral is just a summation, and a sum of quadratic functions is also quadratic. Figure 5.5 shows the basins of attraction for the two weights α and β, created by two clusters in the real line which we defined earlier.

Since the attraction of each cluster-point on each of the two weights depends on their relative position, a graphical representation of the energy function must take both α and β into account. Figure 5.6 is a second visualization

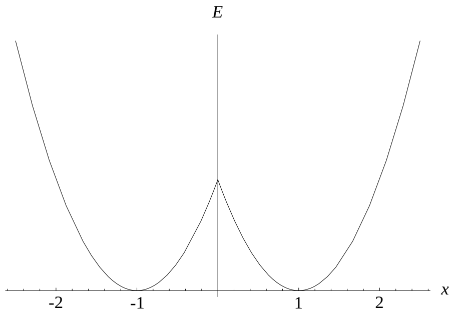


Fig. 5.5. Attractors generated by two one-dimensional clusters

attempt. The vertical axis represents the sum of the distances from α to each one of the cluster-points which lie nearer to α than to β, plus the sum of the distances from β to each one of the cluster-points which lie nearer to β than to α. The gradient of this distance function is proportional to the attraction on α and β. As can be seen there are two global minima: the first at $\alpha = 1, \beta = -1$ and the second at $\alpha = -1, \beta = 1$. Which of these two global minima will be found depends on the weight initialization.

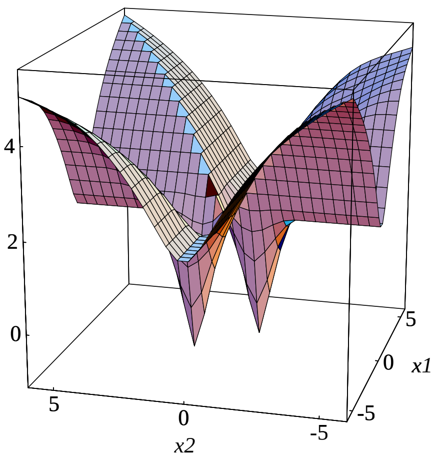


Fig. 5.6. Energy distribution for two one-dimensional clusters

Note, however, that there are also two local minima. If α, for example, is very large in absolute value, this weight will not be updated, so that β converges to 0 and α remains unchanged. The same is valid for β. The dynamics of the learning process is given by *gradient descent* on the energy function. Figure 5.7 is a close-up of the two global minima.

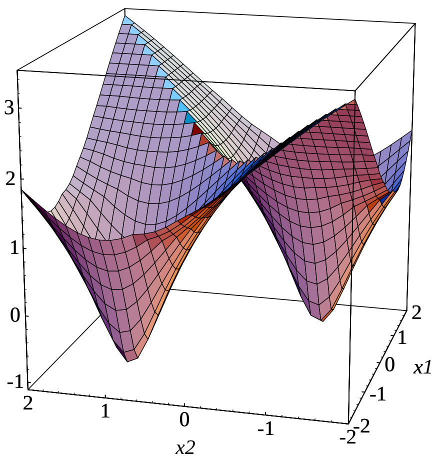


Fig. 5.7. Close-up of the energy distribution

The two local minima in Figure 5.6 correspond to the possibility that unit 1 or unit 2 could become "dead units". This happens when the initial weight vectors lie so far apart from the cluster points that they are never selected for an update.

5.2.2 Multidimensional case – the classical methods

It is not so easy to show graphically how the learning algorithm behaves in the multidimensional case. All we can show are "instantaneous" slices of the energy function which give us an idea of its general shape and the direction of the gradient. A straightforward generalization of the formula used in the previous section provides us with the following definition:

Definition 5. *The energy function of a set* $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$ *of* n*-dimensional normalized vectors* $(n \geq 2)$ *is given by*

$$E_X(\mathbf{w}) = \sum_{i=1}^{m} (\mathbf{x}_i - \mathbf{w})^2$$

where $\mathbf{w}$ *denotes an arbitrary vector in* n*-dimensional space.*

The energy function is the sum of the quadratic distances from $\mathbf{w}$ to the input vectors $\mathbf{x}_i$. The energy function can be rewritten in the following form:

$$E_X(\mathbf{w}) = m\mathbf{w}^2 - 2\sum_{i=1}^{m} \mathbf{x}_i \cdot \mathbf{w} + \sum_{i=1}^{m} \mathbf{x}_i^2$$

$$
\begin{aligned}
&= m(\mathbf{w}^2 - \frac{2}{m}\mathbf{w}\cdot\sum_{i=1}^{m}\mathbf{x}_i) + \sum_{i=1}^{m}\mathbf{x}_i^2 \\
&= m(\mathbf{w} - \frac{1}{m}\sum_{i=1}^{m}\mathbf{x}_i)^2 - \frac{1}{m^2}(\sum_{i=1}^{m}\mathbf{x}_i)^2 + \sum_{i=1}^{m}\mathbf{x}_i^2 \\
&= m(\mathbf{w} - \mathbf{x}^*)^2 + K.
\end{aligned}
$$

The vector $\mathbf{x}^*$ is the centroid of the cluster $\{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_m\}$ and K a constant. The energy function has a global minimum at $\mathbf{x}^*$. Figure 5.8 shows the energy function for a two-dimensional example. The first cluster has its centroid at $(-1, 1)$, the second at $(1, -1)$. The figure is a snapshot of the attraction exerted on the weight vectors when each cluster attracts a single weight vector.

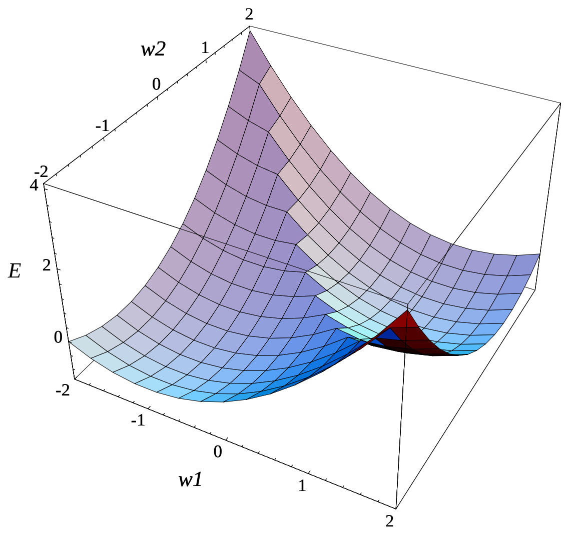


Fig. 5.8. Energy function of two two-dimensional clusters

Statisticians have worked on the problem of finding a good clustering of empirical multidimensional data for many years. Two popular approaches are:

- *k-nearest neighbors* – Sample input vectors are stored and classified in one of ℓ different classes. An unknown input vector is assigned to the class to which the majority of its k closest vectors from the stored set belong (ties can be broken with special heuristics) [110]. In this case a training set is needed, which is later expanded with additional input vectors.
- *k-means* – Input vectors are classified in k different clusters (at the beginning just one vector is assigned to each cluster). A new vector $\mathbf{x}$ is assigned to the cluster k whose centroid $\mathbf{c}_k$ is the closest one to the vector. The centroid vector is updated according to

$$\mathbf{c}_k = \mathbf{c}_k + \frac{1}{n_k}(\mathbf{x} - \mathbf{c}_k),$$

where n_k is the number of vectors already assigned to the k-th cluster. The procedure is repeated iteratively for the whole data set [282]. This method is similar to Algorithm 5.1.1, but there are several variants. The centroids can be updated at the end of several iterations or right after the test of each new vector. The centroids can be calculated with or without the new vector [59].

The main difference between a method like k-nearest neighbors and the algorithm discussed in this chapter is that we do not want to store the input data but only capture its structure in the weight vectors. This is particularly important for applications in which the input data changes with time. The transmission of computer graphics is a good example. The images can be compressed using unsupervised learning to find an adequate *vector quantization*. We do not want to store all the details of those images, only their relevant statistics. If these statistics change, we just have to adjust some network weights and the process of image compression still runs optimally (see Sect. 5.4.2).

5.2.3 Unsupervised learning as minimization problem

In some of the graphics of the energy function shown in the last two sections, we used a strong simplification. We assumed that the data points belonging to a given cluster were known so that we could compute the energy function. But this is exactly what the network should find out. Different assumptions lead to different energy function shapes.

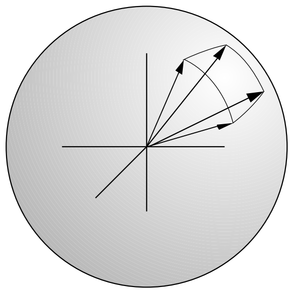


Fig. 5.9. Extreme points of the normalized vectors of a cluster

We can illustrate the iteration process using a three-dimensional example. Assume that two clusters of four normalized vectors are given. One of the clusters is shown in Figure 5.9. Finding the center of this cluster is equivalent to the two-dimensional problem of finding the center of two clusters of four points at the corners of two squares. Figure 5.10 shows the distribution of members of the two clusters and the initial points a and b selected for the iteration process. You can think of these two points as the endpoints of two normalized vectors in three-dimensional space and of the two-dimensional surface as an approximation of the "flattened" surface of the sphere.

The method we describe now is similar to Algorithm 5.1.1, but uses the Euclidian distance between points as metric. The energy of a point in $\mathbb{R}^2$ corresponds to the sum of the quadratic distances to the points in one of the two clusters. For the initial configuration shown in Figure 5.10 all points above the horizontal line lie nearer to a than to b and all points below the line lie nearer to b than to a.

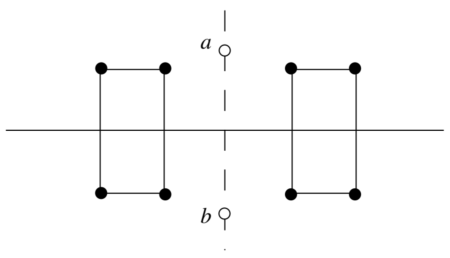


Fig. 5.10. Two cluster and initial points 'a' and 'b'

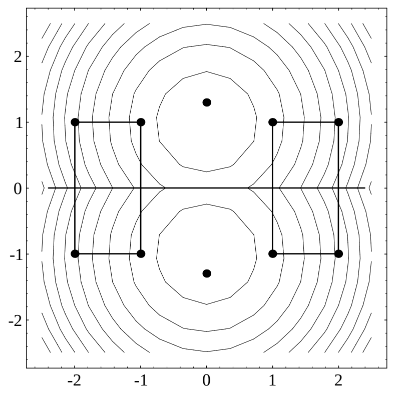


Fig. 5.11. Energy function for the initial points

Figure 5.11 shows the contours of the energy function. As can be seen, both a and b are near to equilibrium. The point a is a representative for the cluster of the four upper points, point b a representative for the cluster of the four lower points. If one update of the position of a and b that modifies the distribution of the cluster points is computed, the shape of the energy function changes dramatically, as shown in Figure 5.12. In this case, the distribution of points nearer to a than to b has changed (as illustrated by the line drawn between a and b). After several more steps of the learning algorithm the situation shown in Figure 5.13 could be reached, which corresponds now to stable equilibrium. The points a and b cannot jump out of their respective clusters, since the iterative corrections always map points inside the squares to points inside the squares.

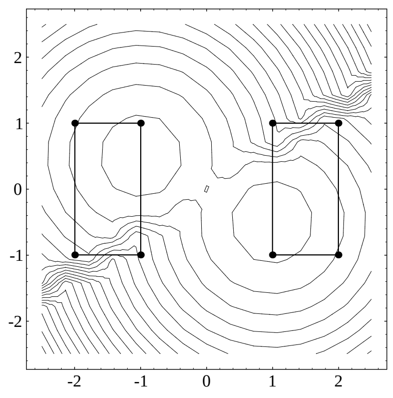


Fig. 5.12. Energy function for a new distribution

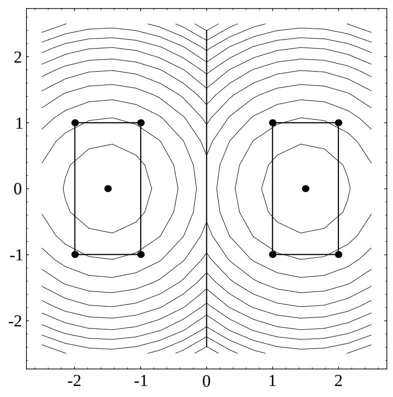


Fig. 5.13. Energy function for the linear separation $x = 0$

5.2.4 Stability of the solutions

The assignment of vectors to clusters can become somewhat arbitrary if we do not have some way of measuring a "good clustering". A simple approach is determining the distance between clusters.

Figure 5.14 shows two clusters of vectors in a two-dimensional space. On the left side we can clearly distinguish the two clusters. On the right side we have selected two weight vectors $\mathbf{w}_1$ and $\mathbf{w}_2$ as their representatives. Each weight vector lies near to the vectors in its cluster, but $\mathbf{w}_1$ lies *inside* the cone defined by its cluster and $\mathbf{w}_2$ *outside*. It is clear that $\mathbf{w}_1$ will not jump outside the cone in future iterations, because it is only attracted by the vectors in its cluster. Weight vector $\mathbf{w}_2$ will at some point jump inside the cone defined by its cluster and will remain there.

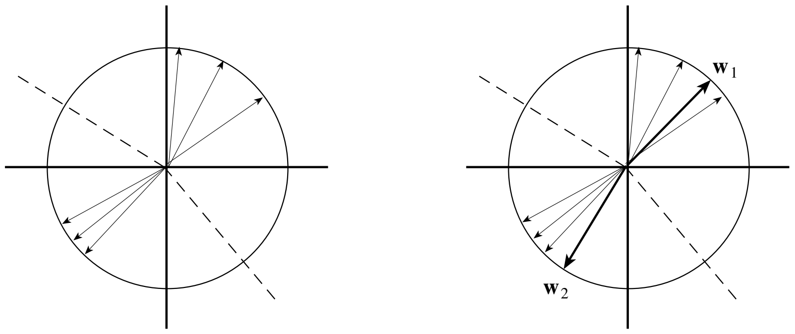


Fig. 5.14. Two vector clusters (left) and two representative weight vectors (right)

This kind of distribution is a *stable solution* or a solution in *stable equilibrium.* Even if the learning algorithm runs indefinitely, the weight vectors will stay by their respective clusters.

As can be intuitively grasped, stable equilibrium requires clearly delimited clusters. If the clusters overlap or are very extended, it can be the case that no stable solution can be found. In this case the distribution of weight vectors remains in *unstable equilibirum.*

Definition 6. *Let P denote the set $\{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_m\}$ of n-dimensional $(n \geq 2)$ vectors located in the same half-space. The* cone K *defined by P is the set of all vectors $\mathbf{x}$ of the form $\mathbf{x} = \alpha_1\mathbf{p}_1 + \alpha_2\mathbf{p}_2 + \cdots + \alpha_m\mathbf{p}_m$, where $\alpha_1, \alpha_2, \ldots, \alpha_m$ are positive real numbers.*

The cone of a cluster contains all vectors "between" the cluster. The condition that all vectors are located in the same half-space forbids degenerate cones filling the whole space.

The *diameter* of a cone defined by normalized vectors is proportional to the maximum possible angle between two vectors in the cluster.

Definition 7. *The* angular diameter φ *of a cone* K*, defined by normalized vectors* $\{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_m\}$ *is*

$$\varphi = \sup\{\arccos(\mathbf{a} \cdot \mathbf{b}) | \forall \mathbf{a}, \mathbf{b} \in K, \text{ with } \|\mathbf{a}\| = \|\mathbf{b}\| = 1\},$$

where $0 \leq \arccos(\mathbf{a} \cdot \mathbf{b}) \leq \pi$*.*

A sufficient condition for stable equilibrium, which formalizes the intuitive idea derived from the example shown in Figure 5.14, is that the angular diameter of the cluster's cone must be smaller than the distance between clusters. This can be defined as follows:

Definition 8. *Let* $P = \{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_m\}$ *and* $N = \{\mathbf{n}_1, \mathbf{n}_2, \ldots, \mathbf{n}_k\}$ *be two non-void sets of normalized vectors in an n-dimensional space* $(n \geq 2)$ *that define the cones* K_P *and* K_N *respectively. If the intersection of the two cones is void, the* angular distance *between the cones is given by*

$$\psi_{PN} = \inf\{\arccos(\mathbf{p} \cdot \mathbf{n}) | \mathbf{p} \in K_P, \mathbf{n} \in K_N, \text{ with } \|\mathbf{p}\| = \|\mathbf{n}\| = 1\},$$

where $0 \leq \arccos(\mathbf{p} \cdot \mathbf{n}) \leq \pi$*. If the two cones intersect, the angular distance between them is zero.*

It is easy to prove that if the angular distance between clusters is greater than the angular diameter of the clusters, a stable solution exists in which the weight vectors of an unsupervised network lie in the cluster's cones. Once there, the weight vectors will not leave the cones (see Exercise 1).

In many applications it is not immediately obvious how to rank different clusterings according to their quality. The usual approach is to define a *cost function* which penalizes too many clusters, and favors less but more compact clusters [77]. An extreme example could be identifying each data point as a cluster. This should be forbidden by the optimization of the cost function.

5.3 Principal component analysis

In this section we discuss a second kind of unsupervised learning and its application for the computation of the *principal components* of empirical data. This information can be used to reduce the dimensionality of the data. If the data was coded using n parameters, we would like to encode them using fewer parameters and without losing any essential information.

5.3.1 Unsupervised reinforcement learning

For the class of algorithms we want to consider we will build networks of *linear associators*. This kind of unit exclusively computes the weighted input as result. This means that we omit the comparison with a threshold. Linear associators are used predominantly in associative memories (Chap. **??**).

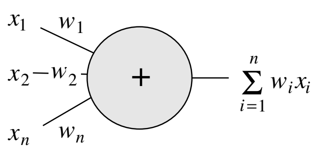


Fig. 5.15. Linear associator

Assume that a set of empirical data is given which consists of n-dimensional vectors $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m\}$. The first principal component of this set of vectors is a vector $\mathbf{w}$ which maximizes the expression

$$\frac{1}{m}\sum_{i=1}^{m} \|\mathbf{w}\cdot\mathbf{x}_i\|^2,$$

that is, the average of the quadratic scalar products. Figure 5.16 shows an example of a distribution centered at the origin (that is, the centroid of the data lies at the origin). The diagonal runs in the direction of maximum variance of the data. The orthogonal projection of each point on the diagonal represents a larger absolute displacement from the origin than each one of its x_1 and x_2 coordinates. It can be said that the projection contains more information than each individual coordinate alone [276]. In order to statistically analyze this data it is useful to make a coordinate transformation, which in this case would be a rotation of the coordinate axis by 45 degrees. The information content of the new x coordinate is maximized in this way. The new direction of the x_1 axis is the direction of the principal component.

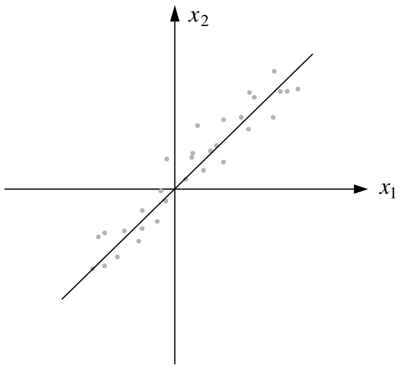


Fig. 5.16. Distribution of input data

The *second* principal component is computed by subtracting from each vector $\mathbf{x}_i$ its projection on the first principal component. The first principal component of the residues is the second principal component of the original data. The second principal component is orthogonal to the first one. The third principal component is computed recursively: the projection of each vector onto the first and second principal components is subtracted from each vector. The first principal component of the residues is now the third principal component of the original data. Additional principal components are computed following such a recursive strategy.

Computation of the principal components makes a reduction of the dimension of the data with minimal loss of information feasible. In the example of Figure 5.16 each point can be represented by a single number (the length of its projection on the diagonal line) instead of two coordinates x_1 and x_2. If we transmit these numbers to a receiver, we have to specify as sender the direction of the principal component and each one of the projections. The transmission error is the difference between the real and the reconstructed data. This difference is the distance from each point to the diagonal line. The first principal component is thus the direction of the line which minimizes the sum of the deviations, that is the optimal fit to the empirical data. If a set of points in three-dimensional space lies on a line, this line is the principal component of the data and the three coordinates can be transformed into a single number. In this case there would be no loss of information, since the second and third principal components vanish. As we can see, analysis of the principal components helps in all those cases in which the input data is concentrated in a small region of the input space.

In the case of neural networks, the set of input vectors can change in the course of time. Computation of the principal components can be done only adaptively and step by step. In 1982 Oja proposed an algorithm for linear associators which can be used to compute the first principal component of empirical data [331]. It is assumed that the distribution of the data is such that the centroid is located at the origin. If this is not the case for a data set, it is always possible to compute the centroid and displace the origin of the coordinate system to fulfill this requirement.

Algorithm 5.3.1 *Computation of the first principal component*

```
start:  Let $X$ be a set of $n$-dimensional vectors.
        The vector $\mathbf{w}$ is initialized randomly ($\mathbf{w} \neq \mathbf{0}$).
        A learning constant $\gamma$ with $0 < \gamma \leq 1$ is selected.
update: A vector $\mathbf{x}$ is selected randomly from $X$.
        The scalar product $\phi = \mathbf{x} \cdot \mathbf{w}$ is computed.
        The new weight vector is $\mathbf{w} + \gamma\phi(\mathbf{x} - \phi\mathbf{w})$.
        Go to update, making $\gamma$ smaller.
```

Of course, a stopping condition has to be added to the above algorithm (for example a predetermined number of iterations). The learning constant γ is chosen as small as necessary to guarantee that the weight updates are not too abrupt (see below). The algorithm is another example of unsupervised learning, since the principal component is found by applying "blind" updates to the weight vector. Oja's algorithm has the additional property of automatically *normalizing* the weight vector $\mathbf{w}$. This saves an explicit normalization step in which we need global information (the value of each weight) to modify each individual weight. With this algorithm each update uses only local information, since each component of the weight vector is modified taking into account only itself, its input, and the scalar product computed at the linear associator.

5.3.2 Convergence of the learning algorithm

With a few simple geometric considerations we can show that Oja's algorithm must converge when a unique solution to the task exists. Figure 5.17 shows an example with four input vectors whose principal component points in the direction of $\mathbf{w}$. If Oja's algorithm is started with this set of vectors and $\mathbf{w}$, then $\mathbf{w}$ will oscillate between the four vectors but will not leave the cone defined by them. If $\mathbf{w}$ has length 1, then the scalar product $\phi = \mathbf{x} \cdot \mathbf{w}$ corresponds to the length of the projection of $\mathbf{x}$ on $\mathbf{w}$. The vector $\mathbf{x} - \phi\mathbf{w}$ is a vector normal to $\mathbf{w}$. An iteration of Oja's algorithm attracts $\mathbf{w}$ to a vector in the cluster. If it can be guaranteed that $\mathbf{w}$ remains of length 1 or close to 1, the effect of a number of iterations is just to bring $\mathbf{w}$ into the middle of the cluster.

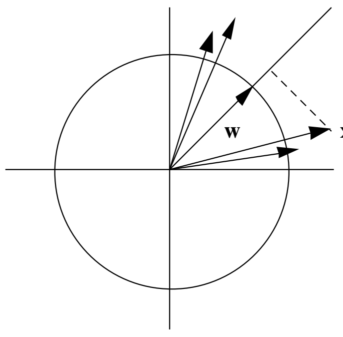


Fig. 5.17. Cluster of vectors and principal component

We must show that the vector $\mathbf{w}$ is automatically normalized by this algorithm. Figure 5.18 shows the necessary geometric constructions. The left side shows the case in which the length of vector $\mathbf{w}$ is greater than 1. Under these circumstances the vector $(\mathbf{x} \cdot \mathbf{w})\mathbf{w}$ has a length greater than the length of the orthogonal projection of $\mathbf{x}$ on $\mathbf{w}$. Assume that $\mathbf{x} \cdot \mathbf{w} > 0$, that is, the vectors $\mathbf{x}$ and $\mathbf{w}$ are not too far away. The vector $\mathbf{x} - (\mathbf{x} \cdot \mathbf{w})\mathbf{w}$ has a negative projection on $\mathbf{w}$ because

$$(\mathbf{x} - (\mathbf{x} \cdot \mathbf{w})\mathbf{w}) \cdot \mathbf{w} = \mathbf{x} \cdot \mathbf{w} - \|\mathbf{w}\|^2 \mathbf{x} \cdot \mathbf{w} < 0.$$

Now we have to think about the result of many iterations of this type. The vector $\mathbf{x} - (\mathbf{x} \cdot \mathbf{w})\mathbf{w}$ has a component normal to $\mathbf{w}$ and another pointing in the negative direction of $\mathbf{w}$. Repeated iterations will bring $\mathbf{w}$ into the middle of the cluster of vectors, so that in the average case the normal components later cancel. However, the negative component does not cancel if $\mathbf{w}$ is still of a length greater than 1. The net effect of the iterations is therefore to position $\mathbf{w}$ correctly, but also to make it smaller. However, care must be taken to avoid making $\mathbf{w}$ *too small* or even reversing its direction in a single iteration. This can be avoided by setting the learning constant γ as small as necessary. It also helps to normalize the training vectors before the algorithm is started. If the vector $\mathbf{x}$ has a positive scalar product ϕ with $\mathbf{w}$, we would like also the new weight vector to have a positive scalar product with $\mathbf{x}$. This means that we want the following inequality to hold

$$\mathbf{x} \cdot (\mathbf{w} + \gamma\phi(\mathbf{x} - \phi\mathbf{w}) > 0.$$

This is equivalent to

$$\gamma(\|\mathbf{x}\|^2 - \phi^2) > -1,$$

and if γ is positive and small the inequality can always be satisfied regardless of the sign of $\|\mathbf{x}\|^2 - \phi^2$.

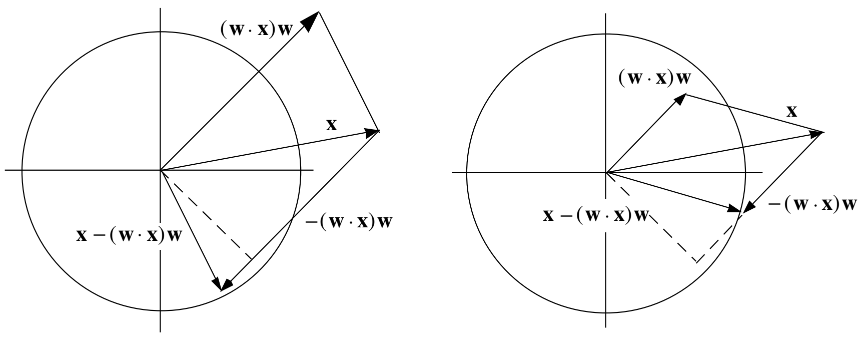


Fig. 5.18. The two cases in Oja's algorithm

The right side of Figure 5.18 shows what happens when the vector $\mathbf{w}$ has a length smaller than 1. In this case the vector $\mathbf{x} - (\mathbf{x} \cdot \mathbf{w})\mathbf{w}$ has a positive projection on $\mathbf{w}$. The normal component of this vector will bring $\mathbf{w}$ into the center of the cluster after repeated iterations. The positive projection of this vector on $\mathbf{w}$ has the net effect of making $\mathbf{w}$ larger. Combining both cases, we can deduce that the net effect of the algorithm is to bring $\mathbf{w}$ into the middle of the cluster, while at the same time the length of $\mathbf{w}$ oscillates around 1 (for a small enough γ). Once this state is achieved, and if the task has a unique solution, $\mathbf{w}$ just oscillates around its equilibrium position and equilibrium length. Note, however, that these are considerations about the *expected value* of **w**. If the clusters are sparsely populated or if the data vectors are of very different lengths, the actual value of **w** can differ significantly at each iteration from its average value.

It can be shown that the first principal component of a set of vectors is equivalent to the direction of the longest eigenvector of their correlation matrix [225] and that Oja's algorithm finds approximately this direction.

5.3.3 Multiple principal components

Sanger proposed a network architecture capable of finding the first m principal components of a data set [387]. The idea is to compute the first principal component using a linear associator and then to subtract the projection of the data on this direction from the data itself. The residues are processed by the next linear associator in the chain. This unit computes the second principal component of the original data. Figure 5.19 shows the structure of the network. The figure shows the connections associated with vector $\mathbf{w}_1$ as an arrow in order to simplify the wiring diagram.

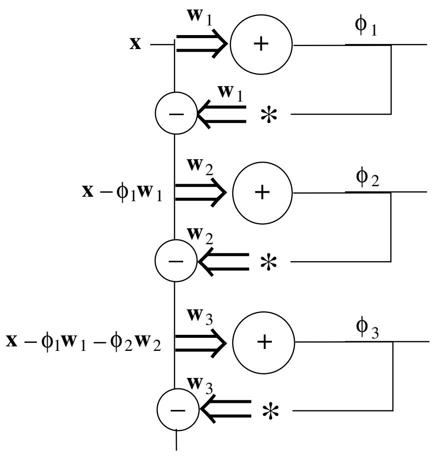


Fig. 5.19. Network for the iterative computation of the first three principal components

The network weights are found using Oja's algorithm at each level of the network. It must be guaranteed that the individual weight vectors (which appear twice in the network) are kept consistent. The three linear associators shown in the figure can be trained simultaneously, but we can stop training the last unit only after the weight vectors of the previous units have stabilized.

Other authors have proposed alternative network topologies to compute the first m principal components of empirical data [332, 381], but the essential idea is roughly the same.

5.4 Some applications

Vector quantization and unsupervised clustering have found many interesting applications. In this section we discuss two of them.

5.4.1 Pattern recognition

Unsupervised learning can be used in all cases where the number of different input clusters is known *a priori*, as is the case with optical character recognition. Assume that letters in a document are scanned and centered in 16×16 projection screens. Each letter is coded as a vector of dimension 256, assigning the value 1 to black pixels and -1 to white pixels. There is some noise associated with each image, so that sometimes pixels are assigned the false bit. If the number of errors per image is not excessive, then the input vectors for the same letter cluster around a definite region in input space. These clusters can be identified by a network of competitive units.

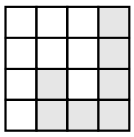


Fig. 5.20. A digital "J" (4×4 matrix)

If only 26 different letters are to be recognized, our network needs only 26 different units. The weights of each unit can be initialized randomly and an unsupervised learning algorithm can be used to find the correct parameters. There are conflicts with letters having a similar shape (for example O and Q), which can be solved only by pulling the clusters apart using a higher scanning resolution.

5.4.2 Image compression

Assume that a picture with 1024×1024 pixels is to be transmitted. The number of bits needed is 2^{20}. However, images are not totally random. They have a structure which can be analyzed with the purpose of compressing the image before transmission. This can be done by dividing the picture in 128×128 fields of 8×8 pixels. Each field can contain any of 2^{64} different possible patterns, but assume that we decide to classify them in 64 different classes. We start an unsupervised network with 64 outputs and train it to classify all 8×8 patterns found in the image. Note that we do not train the network with all 2^{64} different possible patterns, but only using those that actually appear in the picture. The units in the network classify these patterns in clusters. The weight vectors of the 64 units are then transmitted as a representative of each cluster and correspond to a certain pattern identified by the corresponding unit. The 64 weight vectors are the *codebook* for the transmission.

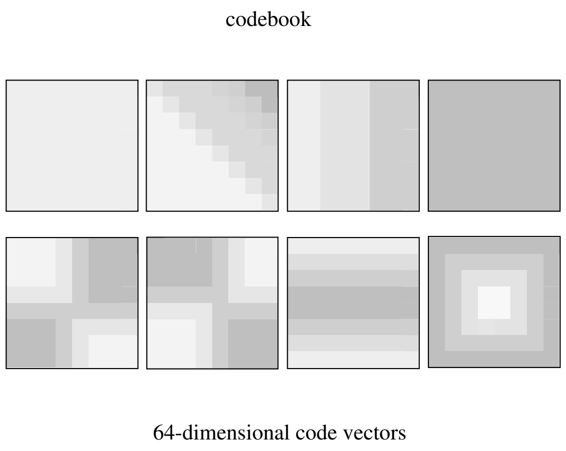


Fig. 5.21. Part of a codebook for image compression

The transmission of the image can begin once the sender and receiver have agreed on a codebook. For each of the 128×128 fields in the picture, the sender transmits the name of the code vector nearer to the 8×8 content of the field. Since there are only 64 codebook vectors, the sender has to transmit 6 bits. The compression ratio achieved for the transmission is $64/6 = 10.66$. The image reconstructed by the sender contains some errors, but if a good clustering has been found, those errors will be minimal. Figure 5.21 shows 8 of the weight vectors found by unsupervised learning applied to a picture. The weight vectors are represented by gray values and show the kind of features identified by the algorithm. Note that there are weight vectors for units which recognize vertical or horizontal lines, white or black regions as well as other kinds of structures. The compression ratio can be modified by using more or less codebook vectors, that is, more or less clusters.

Scientists have long speculated about the possible mechanisms of information compression in the human brain. Many different experiments have shown that the connection pattern of the brain is determined by genetic factors, but also by experience. There is an interplay between "nature and nurture" that leads to the gradual modification of the neural connections. Some classical results were obtained analyzing the visual cortex of cats. It is now well known that if a cat is blind in one eye, the part of the visual cortex normally assigned to that eye is reassigned to the other one. Some other experiments, in which cats were kept in a special environment (lacking for example any kind of horizontal lines) have shown that feature detectors in the brain specialize to identify only those patterns present in the real world. These results are interpreted by assuming that genetic factors determine the *rules* by which unsupervised learning takes place in the neural connections, whereas actual experience shapes the feature detectors. Some regions of the visual cortex are in fact detectors which identify patterns of the kind shown in Figure 5.21.

5.5 Historical and bibliographical remarks

Frank Rosenblatt was the first to experiment with unsupervised learning in networks of threshold elements [382]. He interpreted the ability of the networks to find clusters as a kind of mechanism similar to the abstraction power of the human mind, capable of forming new concepts out of empirical material.

After Rosenblatt there were not many other examples of unsupervised networks until the 1970s. Grossberg and von der Malsburg formulated biologically oriented models capable of self-organizing in an unsupervised environment [168, 284]. Theoretical models of synaptic plasticity were developed later by Sejnowski and Tesauro [397] and Klopf [249]. The Nobel prizewinner Gerald Edelman developed a whole theory of "neuronal selection" with which he tried to explain the global architecture of the brain [124]. Using large computer simulations he showed that the "market forces", that is, competition at all levels of the neuronal circuits, can explain some of their characteristics.

Vector quantization has become a common tool for the transmission and analysis of images and signals. Although the resulting assignment problem is known to be computationally expensive (*NP*-hard indeed, see Chap. 10), many heuristics and approximate methods have been developed that are now being used [159]. Codebooks have also been applied to the digital transmission of voice signals over telephone lines.

Principal component analysis has been a standard statistical technique since it was first described by Pearson and Hotelling between 1901 and 1903 [225]. The methods were discussed analytically but only found real application with the advent of the computer. Traditional methods of principal component analysis proceed by finding eigenvalues of the correlation matrix. Adaptive methods, of the kind introduced in the 1970s, do not assume that the data set is fixed and constant, but that it can change in the course of time [387]. The algorithms of Oja and Linsker can be used in these cases. The possible applications of such an adaptive principal component analysis are real-time compression of signals, simplification and acceleration of learning algorithms for neural networks [404], or adaptive pattern recognition [208].

Exercises

1. Prove that if m weight vectors lie in the cones of m clusters such that their angular diameters are smaller that the minimum angular distance

between clusters, unsupervised learning will not bring any of the weight vectors out of its respective cone.

2. Implement an optical character recognizer. Train a classifier network of competitive units with scanned examples of 10 different letters. Test if the network can identify noisy examples of the same scanned letters.
3. Train the classifier network using Oja's algorithm.
4. How can dead units be avoided in a network of competitive units? Propose two or three different heuristics.
5. Compute a codebook for the compression of a digitized image. Compare the reconstructed image with the original. Can you improve the reconstructed image using a Gaussian filter?
6. A network can be trained to classify characters using a decision tree. Propose an architecture for such a network and a training method.

6

One and Two Layered Networks

6.1 Structure and geometric visualization

In the previous chapters the computational properties of isolated threshold units have been analyzed extensively. The next step is to combine these elements and look at the increased computational power of the network. In this chapter we consider feed-forward networks structured in successive layers of computing units.

6.1.1 Network architecture

The networks we want to consider must be defined in a more precise way in terms of their *architecture*. The atomic elements of any architecture are the computing units and their interconnections. Each computing unit collects the information from n input lines with an *integration function* $\Psi : \mathbb{R}^n \to \mathbb{R}$. The total excitation computed in this way is then evaluated using an *activation function* $\Phi : \mathbb{R} \to \mathbb{R}$. In perceptrons the integration function is the sum of the inputs. The activation (also called output function) compares the sum with a threshold. Later we will generalize Φ to produce all values between 0 and 1. In the case of Ψ some functions other than addition can also be considered [454], [259]. In this case the networks can compute some difficult functions with fewer computing units.

Definition 9. *A network architecture is a tuple (I, N, O, E) consisting of a set I of input sites, a set N of computing units, a set O of output sites and a set E of weighted directed edges. A directed edge is a tuple (u, v, w) whereby $u \in I \cup N$, $v \in N \cup O$ and $w \in \mathbb{R}$.*

The input sites are just entry points for information into the network and do not perform any computation. Results are transmitted to the output sites. The set N consists of all computing elements in the network. Note that the edges between all computing units are weighted, as are the edges between input and output sites and computing units.

In neural network literature there is an inconsistency in notation that unfortunately has become tradition. The input sites of a network are usually called input units, although nothing is computed here. The output sites of the network are implicit in the construction but not explicitly given. The computing units from which results are read off are called the output units.

Layered architectures are those in which the set of computing units N is subdivided into ℓ subsets $N_1, N_2, \ldots, N_\ell$ in such a way that only connections from units in N_1 go to units in N_2, from units in N_2 to units in N_3, etc. The input sites are only connected to the units in the subset N_1, and the units in the subset N_ℓ are the only ones connected to the output sites. In the usual terminology, the units in N_ℓ are the output units of the network. The subsets N_i are called the *layers* of the network. The set of input sites is called the *input layer*, the set of output units is called the *output layer*. All other layers with no direct connections from or to the outside are called *hidden layers*. Usually the units in a layer are not connected to each other (although some neural models make use of this kind of architecture) and the output sites are omitted from the graphical representation.

A neural network with a layered architecture does not contain cycles. The input is processed and relayed from one layer to the other, until the final result has been computed. Figure 6.1 shows the general structure of a layered architecture.

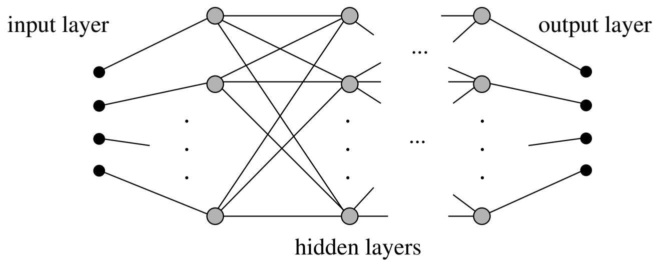


Fig. 6.1. A generic layered architecture

In layered architectures normally all units from one layer are connected to all other units in the following layer. If there are m units in the first layer and n units in the second one, the total number of weights is mn. The total number of connections can become rather large and one of the problems with which we will deal is how to reduce the number of connections, that is, how to *prune* the network.

6.1.2 The XOR problem revisited

The properties of one- and two-layered networks can be discussed using the case of the XOR function as an example. We already saw that a single perceptron cannot compute this function, but a two-layered network can. The

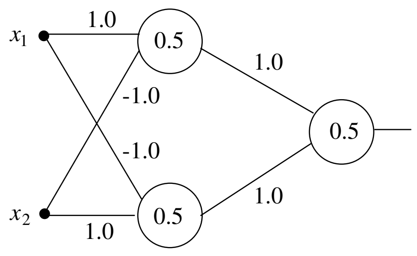


Fig. 6.2. A three-layered network for the computation of XOR

network in Figure 6.2 is capable of doing this using the parameters shown in the figure. The network consists of three layers (adopting the usual definition of the layer of input sites as input layer) and three computing units. One of the units in the hidden layer computes the function $x_1 \wedge \neg x_2$, and the other the function $\neg x_1 \wedge x_2$. The third unit computes the OR function, so that the result of the complete network computation is

$$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2).$$

A natural question to ask is how many basically different solutions can be implemented with this network. Different solutions are only those expressed with a different combination of two-dimensional binary functions, not those which differ only in the weights being used and not in the individual functions being computed. The question then is how many different expressions for XOR can be written using only three out of 14 of the 16 possible Boolean functions of two variables (since XOR and ¬XOR are not among the possible building blocks). An exhaustive search for all possible combinations can be made and the solution is the one shown in Figure 6.3.

The notation used in the figure is as follows: since we are considering logical functions of two variables, there are four possible combinations for the input. The outputs for the four inputs are four bits which uniquely distinguish each logical function. We use the number defined by these four bits as a subindex for the name of the functions. The function $(x_1, x_2) \mapsto 0$, for example, is denoted by f_0 (since 0 corresponds to the bit string 0000). The AND function is denoted by f_8 (since 8 corresponds to the bit string 1000), whereby the output bits are ordered according to the following ordering of the inputs: (1,1), (0,1), (1,0), (0,0).

The sixteen possible functions of two variables are thus:

$$\begin{aligned}
f_0(x_1,x_2) &= f_{0000}(x_1,x_2) = 0\\
f_1(x_1,x_2) &= f_{0001}(x_1,x_2) = \neg(x_1 \vee x_2)\\
f_2(x_1,x_2) &= f_{0010}(x_1,x_2) = x_1 \wedge \neg x_2\\
f_3(x_1,x_2) &= f_{0011}(x_1,x_2) = \neg x_2\\
f_4(x_1,x_2) &= f_{0100}(x_1,x_2) = \neg x_1 \wedge x_2\\
f_5(x_1,x_2) &= f_{0101}(x_1,x_2) = \neg x_1\\
f_6(x_1,x_2) &= f_{0110}(x_1,x_2) = x_1 \oplus x_2\\
f_7(x_1,x_2) &= f_{0111}(x_1,x_2) = \neg(x_1 \wedge x_2)
\end{aligned}$$

$$\begin{aligned}
f_8(x1,x_2) &= f_{1000}(x_1,x_2) = x_1 \wedge x_2\\
f_9(x1,x_2) &= f_{1001}(x_1,x_2) = x_1 \equiv x_2\\
f_{10}(x1,x_2) &= f_{1010}(x_1,x_2) = x_1\\
f_{11}(x1,x_2) &= f_{1011}(x_1,x_2) = x_1 \vee \neg x_2\\
f_{12}(x1,x_2) &= f_{1100}(x_1,x_2) = x_2\\
f_{13}(x1,x_2) &= f_{1101}(x_1,x_2) = \neg x_1 \vee x_2\\
f_{14}(x1,x_2) &= f_{1110}(x_1,x_2) = x_1 \vee x_2\\
f_{15}(x1,x_2) &= f_{1111}(x_1,x_2) = 1
\end{aligned}$$

Figure 6.3 shows all solutions found by an exhaustive search. The network of Figure 6.2 corresponds to the function composition

$$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2) = f_{14}(f_2(x_1,x_2), f_4(x_1,x_2)).$$

Increasing the number of units in the hidden layer increases the number of possible combinations available. We say that the *capacity* of the network increases. Note the symmetry in the function compositions of Figure 6.3, which is not just a random effect as we show later on.

The network of Figure 6.4 can also be used to compute the XOR function. This is not a pure layered architecture, since there are direct connections from the input sites to the output unit. The output unit computes the OR function of the two inputs but is inhibited by the first unit if both inputs are 1.

6.1.3 Geometric visualization

The symmetry of the 16 basic solutions for the XOR problem can be understood by looking at the regions defined in weight space by the two-layered network. Each of the units in Figure 6.2 separates the input space into a closed positive and an open negative half-space. Figure 6.5 shows the linear separations defined by each unit and the unit square. The positive half-spaces have been shaded.

The three regions defined in this way can be labeled with two bits: the first bit is 1 or 0 according to whether this region is included in the positive or negative half-space of the first linear separation. The second bit is 1 or 0 if it is included in the positive or negative half-space of the second linear separation. In this way we get the labeling shown in Figure 6.6.

The two units in the first layer produce the labeling of the region in which the input is located. The point $(1,1)$, for example is contained in the region 00.

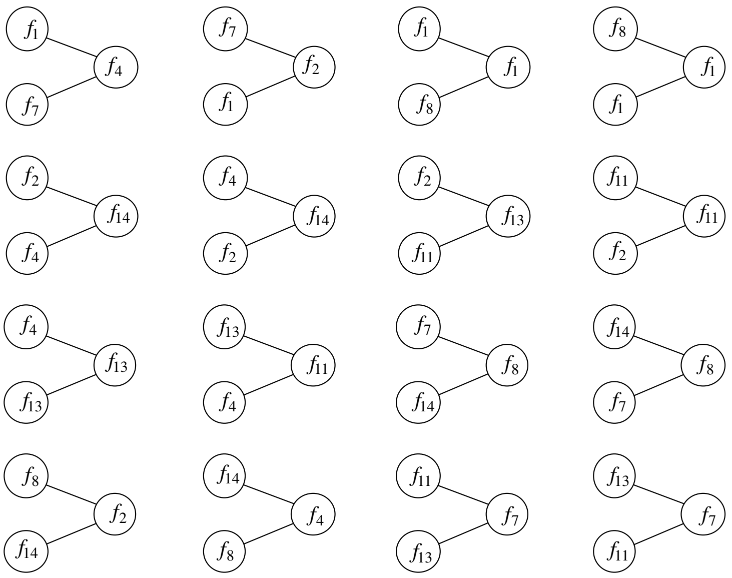


Fig. 6.3. The 16 solutions for the computation of XOR with three computing units

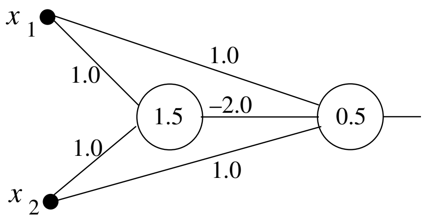


Fig. 6.4. Two unit network for the computation of XOR

This recoding of the input bits makes the XOR problem solvable, because the output unit must only decode three region labels. Only the shaded areas must produce a 1 and this can be computed with the OR function applied to the two bits of the regions labels. This is a general feature of layered architectures: the first layer of computing units maps the input vector to a second space, called classification or feature space. The units in the last layer of the network must decode the classification produced by the hidden units and compute the final output.

We can now understand in a more general setting how layered networks work by visualizing in input space the computations they perform. Each unit in the first hidden layer computes a linear separation of input space. Assume that input space is the whole of $\mathbb{R}^2$. It is possible to isolate a well-defined cluster of points in the plane by using three linear separations as shown in Figure 6.7. Assume that we are looking for a network which computes the

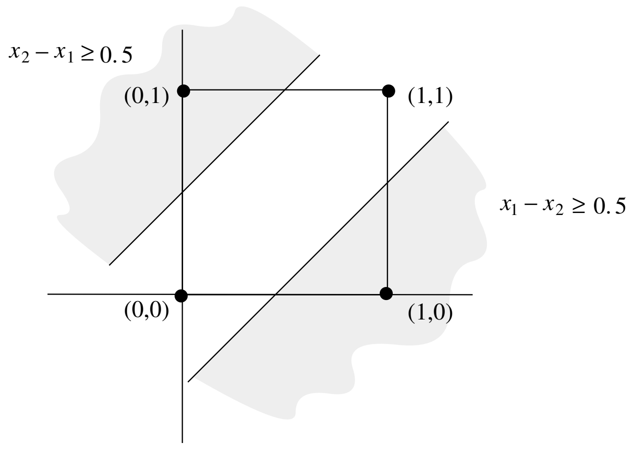


Fig. 6.5. Space separation defined by a two-layered network

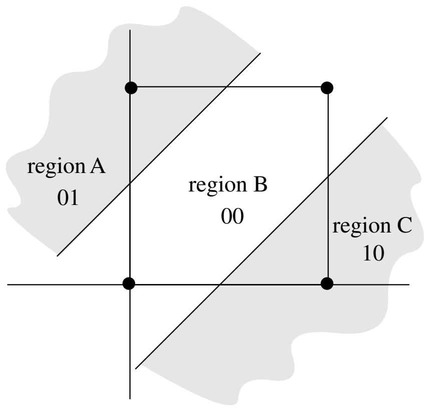


Fig. 6.6. Labeling of the regions in input space

value 1 for the points in the cluster. Three hidden units, and an output unit which computes the AND function of three inputs, can solve this problem. The output unit just decodes the label of the shaded region (111) and produces in this case a 1. Note that, in general, to define a convex cluster in an input space of dimension n at least $n + 1$ hidden units are needed.

If the union of two clusters has to be identified and points in them are assigned the value 1, it is possible to use three units in the first hidden layer to enclose the first cluster and another three units in this layer to enclose the second cluster. Two AND units in the second hidden layer can identify when a point belongs to one or to the other cluster. A final output unit computes the OR function of two inputs. Such a network can identify points in the union of the two clusters. In general, any union of convex polytopes in input space can be classified in this way: units in the first hidden layer define the sides of

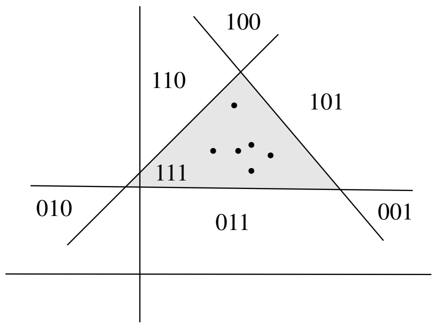


Fig. 6.7. Delimiting a cluster with three linear separations

the polytopes, the units in the second layer the conjunction of sides desired, and the final output unit computes whether the input is located inside one of the convex polytopes.

6.2 Counting regions in input and weight space

The construction used in the last section to isolate clusters is not optimal, because no effort is made to "reuse" hyperplanes already defined. Each cluster is treated in isolation and uses as many units as necessary. In general we do not know how many different clusters are contained in the data and besides the clusters do not need to be convex. We must look more deeply into the problem of how many regions can be defined by intersecting half-spaces and why in some cases a network does not contain enough "plasticity" to solve a given problem.

6.2.1 Weight space regions for the XOR problem

Assume that we are interested in finding the weight vectors for a perceptron capable of computing the AND function. The weights w_1, w_2, w_3 must fulfill the following inequalities:

$$\begin{aligned}
&\text{for the point (0,0):} 0 \cdot w_1 + 0 \cdot w_2 + 1 \cdot w_3 < 0, \text{output} = 0\,,\\
&\text{for the point (0,1):} 0 \cdot w_1 + 1 \cdot w_2 + 1 \cdot w_3 < 0, \text{output} = 0\,,\\
&\text{for the point (1,0):} 1 \cdot w_1 + 0 \cdot w_2 + 1 \cdot w_3 < 0, \text{output} = 0\,,\\
&\text{for the point (1,1):} 1 \cdot w_1 + 1 \cdot w_2 + 1 \cdot w_3 \geq 0, \text{output} = 1\,.
\end{aligned}$$

These three inequalities define half-spaces in three-dimensional weight space. The four separating planes go through the origin and are given by the equations:

$$\begin{aligned}
&\text{plane 1:}\, w_3 = 0\\
&\text{plane 2:}\, w_2 + w_3 = 0\\
&\text{plane 3:}\, w_1 + w_3 = 0\\
&\text{plane 4:}\, w_1 + w_2 + w_3 = 0
\end{aligned}$$

Three separating planes in a three-dimensional space define 8 different regions, but four separating planes define only 14 regions. Each region corresponds to one of 14 possible combinations of inequality symbols in the set of four inequalities which defines a Boolean function. Since there are 16 Boolean functions of two variables, two of them cannot be computed with a perceptron. We already know that they are the XOR and ¬XOR functions.

We can visualize the fourteen regions with the help of a three-dimensional sphere. It was shown in Chapter 4 that the four inequalities associated with the four points (1,1), (0,1), (1,0), and (0,0) define a solution polytope in weight space. One way of looking at the regions defined by m hyperplane cuts going through the origin in an n-dimensional space is by requiring that the weight vectors for our perceptron be normalized. This does not affect the perceptron computation and is equivalent to the condition that the tip of all weight vectors should end at the unit hypersphere of dimension n. In this way all the convex regions produced by the m hyperplane cuts define solution regions on the "surface" of the unit sphere. The 14 solution polytopes define 14 solution regions. Each region corresponds to a logical function. Figure 6.8 shows some of them and their labeling. Each label consists of the four output bits for the four possible binary inputs associated with the function. The region 0000, for example, is the solution region for the function $f_{0000} = f_0$. The region 1000 is the solution region for the function $f_{1000} = f_8$, i.e., the AND function.

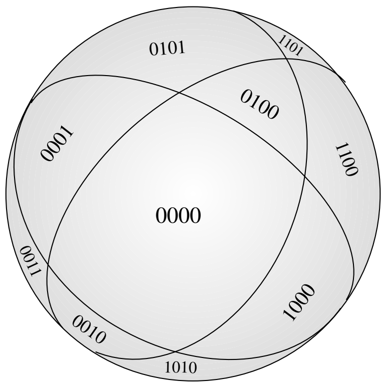


Fig. 6.8. The Boolean sphere

The labeling of neighboring regions separated by a great circle differs in just one bit, as is clear from Figure 6.9. The only regions present are delimited by three of four circles. The AND region (1000) has only the three neighbors

0000, 1100 and 0011, because the region 1001 is void. It corresponds to a non-computable function.

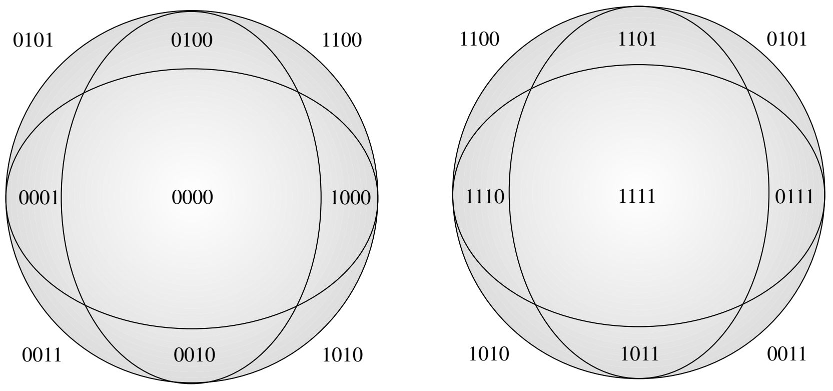


Fig. 6.9. Two opposite sides of the Boolean sphere

6.2.2 Bipolar vectors

Many models of neural networks use bipolar, not binary, vectors. In a bipolar coding the value 0 is substituted by −1. This change does not affect the essential properties of the perceptrons, but changes the symmetry of the solution regions. It is well known that the algebraic development of some terms useful for the analysis of neural networks becomes simpler when bipolar coding is used.

With a bipolar coding the equations for the 4 cutting planes of the three-dimensional Boolean sphere become

$$\begin{aligned}\text{plane 1:} & -w_1 - w_2 + w_3 = 0\\ \text{plane 2:} & -w_1 + w_2 + w_3 = 0\\ \text{plane 3:} & \quad w_1 - w_2 + w_3 = 0\\ \text{plane 4:} & \quad w_1 + w_2 + w_3 = 0\end{aligned}$$

All three planes meet at the origin and form symmetric solution polytopes, since the vectors normal to the planes have pairwise scalar products of 1 or −1.

Since the relative sizes of the solution regions on the Boolean sphere represent how difficult it is to learn them, and since our learning algorithm will be asked to learn one of these functions randomly, the best strategy is to try to get regions of about the same relative size. Table 6.1 was calculated using a Monte Carlo method. A normalized weight vector was generated randomly and its associated Boolean function was computed. By repeating the experiment a number of times it was possible to calculate the relative volumes of the solution regions. The table shows that the maximum variation in the relative sizes of the 14 possible regions is given by a factor of 1.33 when bipolar coding is used, whereas in the binary case it is about 12.5. This means that with binary coding some regions are almost one order of magnitude smaller than others. And indeed, it has been empirically observed that multilayer neural networks are easier to train using a bipolar representation than a binary one [341]. The rationale for this is given by the size of the regions in the Boolean sphere. It is also possible to show that bipolar coding is optimal under this criterion.

Table 6.1. Relative sizes of the regions on the Boolean sphere as percentage of the total surface

Coding	*Boolean function number*							
	0	1	2	3	4	5	6	7
binary	26.83	2.13	4.18	4.13	4.17	4.22	0.00	4.13
bipolar	8.33	6.29	6.26	8.32	6.24	8.36	0.00	6.22
Coding	*Boolean function number*							
	8	9	10	11	12	13	14	15
binary	4.28	0.00	4.26	4.17	4.17	4.14	2.07	27.12
bipolar	6.16	0.00	8.42	6.33	8.27	6.31	6.25	8.23

Bipolar coding is still better in n-dimensional space and for a large n, since two randomly selected vectors with coordinates 1 or -1 are orthogonal with a probability near 1. This is so because each component is 1 or -1 with probability 1/2. The expected value of the scalar product is small compared to the length of the two vectors, which is $\sqrt{n}$. Since the separating planes in weight space are defined by these vectors, they also tend to be mutually orthogonal when bipolar coding is used. The expected value of the scalar product of binary vectors, on the other hand, is $n/4$, which is not negligible when compared to $\sqrt{n}$, even for large n.

6.2.3 Projection of the solution regions

Another way of looking at the solution regions in the surface of the Boolean sphere is by projecting them onto a plane. A kind of stereographic projection can be used in this case. The stereographic projection is shown in Figure 6.10. From the north pole of the sphere a line is projected to each point P on the surface of the sphere and its intersection with the plane is the point in the plane associated with P. This defines a unique mapping from the surface of the sphere to the plane, adopting the convention that the north pole itself is mapped to a point at infinity.

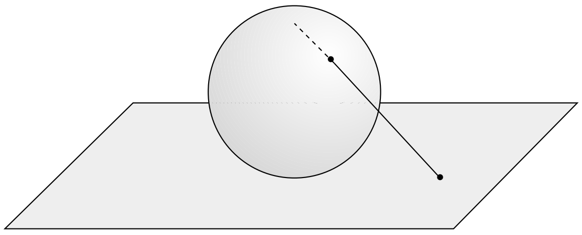


Fig. 6.10. Stereographic projection

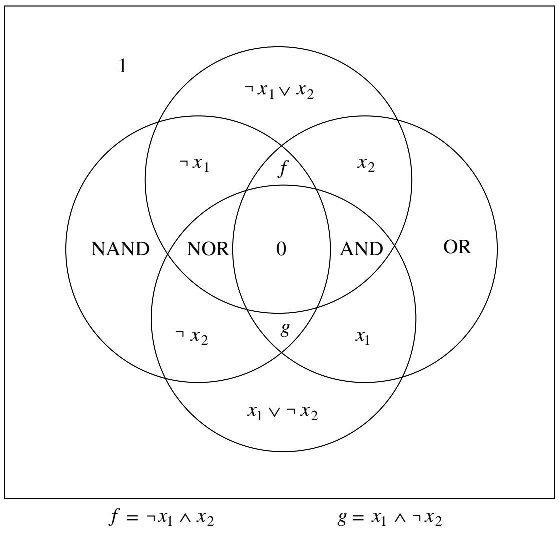


Fig. 6.11. Projection of the solution regions of the Boolean sphere

The stereographic projection projects circles on the sphere (which do not touch the north pole) to ellipses. The four cuts produced by the four separating hyperplanes define four circles on the surface of the Boolean sphere, and these in turn four ellipses on the projection plane. Since we are not interested in the exact shape of these projections, but in the regions they define on the plane, we transform them into circles. Figure 6.11 shows the result of such a projection when the center of region 1111 is chosen as the north pole.

Instead of repeating the four-bit labeling of the regions in Figure 6.11 the expressions for the logical functions “contained” in each region are written explicitly. It is obvious that the number of regions cannot be increased, because four circles on a plane cannot define more than 14 different regions. This result is related to the *Euler characteristic* of the plane [63].

The symmetry of the solution regions is made more evident by adopting a stylized representation. Only the neighborhood relations are important for

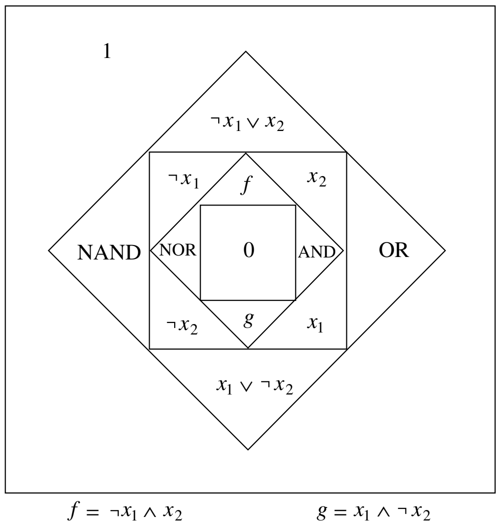


Fig. 6.12. Stylized representation of the projected solution regions

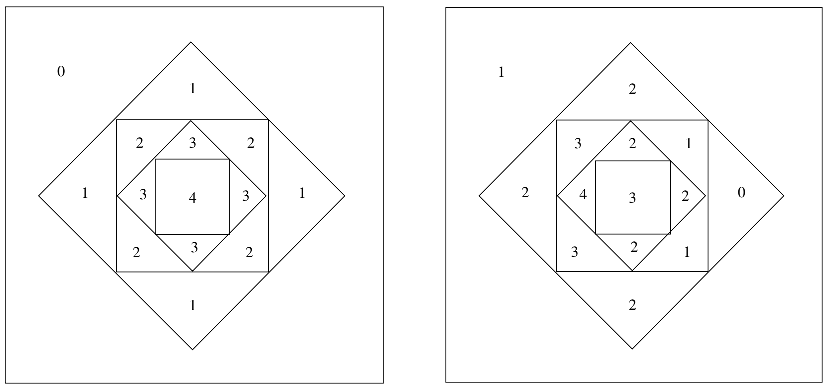


Fig. 6.13. Error functions for the computation of f_{1111} and $x_1 \vee x_2$

our discussion, so we transform Figure 6.11 into Figure 6.12. We can see that functions f and $\neg f$ are always located in symmetrical regions of the space.

The number of neighbors of each region is important if an iterative algorithm is used which, starting from a randomly chosen point, goes from one region to another, trying to minimize the classification error, as the perceptron learning algorithm does. The error function for a given Boolean function of two variables can be represented with this kind of diagram. Figure 6.13 shows the values of the error in each region when looking for the parameters to compute the function f_{1111}.

There is a global maximum and a global minimum only. The first is the solution region for f_{0000}, the second, the solution region for f_{1111}. Starting from a randomly selected point, a possible strategy is to greedily descend the error function. From each region with error 1 there is one path leading to a region with error 0. From regions with error two, there are two alternatives and from regions with error three, three possible paths.

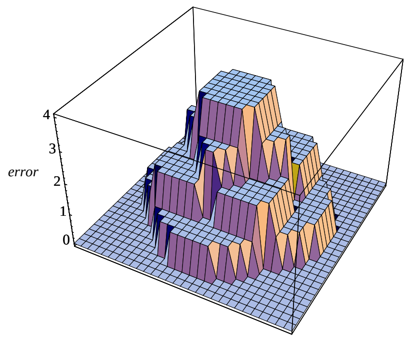


Fig. 6.14. The error function for f_{1111}

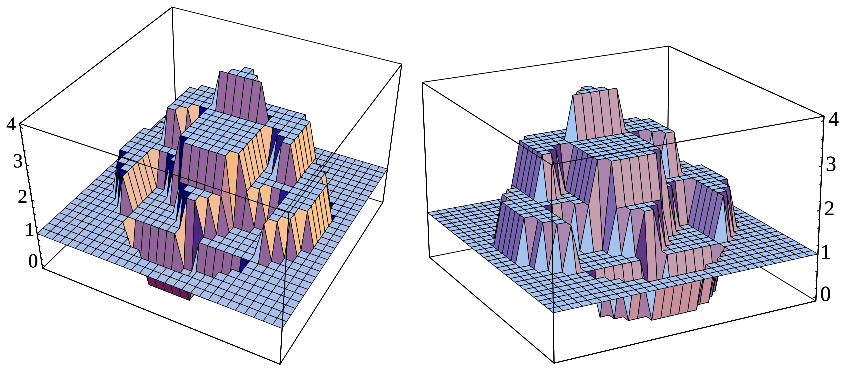


Fig. 6.15. Two perspectives of the error function for OR

The error distribution in weight space in shown in Figure 6.13. The global maximum lies in the solution region for ¬XOR. From each region there is a path which leads to the global minimum.

Figures 6.14 and 6.15 show the error functions for the functions f_{1111} and f_{1110} (OR). The global maxima and minima can be readily identified.

6.2.4 Geometric interpretation

We analyzed the surface of the Boolean sphere in so much detail because it gives us a method to interpret the functioning of networks with a hidden layer. This is the problem we want to analyze now.

Consider a network with three perceptrons in the hidden layer and one output unit. An input vector is processed and the three hidden units produce a new code for it using three bits. This new code is then evaluated by the output unit. Each unit in the hidden layer separates input space into two half-spaces. In order to simplify the visualization, we only deal with normalized vectors. Each division of input space is equivalent to a subdivision of the unit sphere which now represents vectors in input space. Figure 6.16 shows a stylized graphical representation of this idea. The input vector has three components.

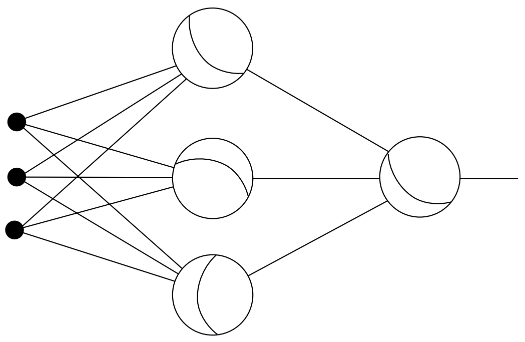


Fig. 6.16. Stylized representation of the input space separations

The separation of input space can be summarized in a single unit sphere in three-dimensional space (Figure 6.17). The three units in the hidden layer produce the labeling of three bits for each region on the sphere. The output unit decodes the three bits and, according to the region, computes a 1 or a 0.

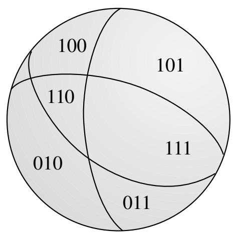


Fig. 6.17. Labeling of the regions in input space

This kind of representation leads to an important idea: the “shattering” of an input space by a class of concepts. In computational learning theory we are interested in dealing with elements from an input space which can be arranged into subsets or classes. If a subset S of an input space X is given, and its points are assigned the value 1 or 0, we are interested in determining the “concept” which can correctly classify this subset of X. In the case of perceptrons the input space is $\mathbb{R}^n$ and the concepts are half-spaces. If the positive elements of subset S (the elements with associated value 1) are located in one half-space, then it is said to be learnable, because one of our concepts (i.e., a half-space) can correctly classify the points of S. One important question is, what is the maximum number of elements of an input space which can be classified by our concepts. In the case of perceptrons with two inputs, this number is three. We can arrange three points arbitrarily in $\mathbb{R}^2$ and assign each one a 0 or a 1, and there is always a way to separate the positive from the negative examples. But four points in general position cannot be separated and the XOR function illustrates this fact.

What kind of shatterings (divisions of input space) are produced by a network with two units in the hidden layer? This question is easier to answer by considering the surface of the unit sphere in input space. In general the two units in the hidden layer divide the surface of the sphere into four regions. The output unit assigns a 1 or a 0 to each region. Figure 6.18 shows the possible shatterings. Regions in which the input is assigned the value 1 have been shaded and regions in which the input is mapped to zero are shown in white. There are sixteen possible colorings, but two of them are impossible because the output unit cannot decode the XOR function. Our network can only produce 14 different shatterings.

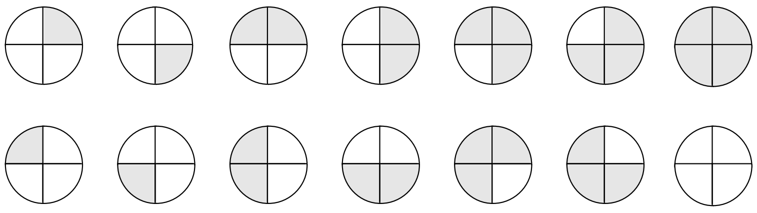


Fig. 6.18. Coloring of the regions in input space

The XOR problem can be solved with one of the shatterings of Figure 6.18 and, in general, any four points in input space can be divided arbitrarily into a positive and a negative class using two hidden units (two dividing lines). However, eight points in input space cannot be divided using only two lines. Consider the example of Figure 6.19. In this case the points at the corners of the square belong to the positive class, the points in the middle of each edge to the negative class. It is not difficult to see that no combination of two separating lines can divide input space in such a way as to separate both classes. In this case we say that the shattering produced by the class of concepts represented by our network does not cover all possible subsets of eight points and not every Boolean function defined on these eight points is *learnable.*

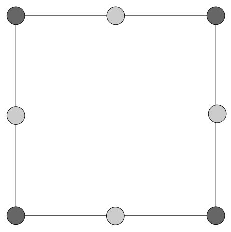


Fig. 6.19. Example of a non-learnable concept for two linear separations

The maximum number of points d of an input space X which can be shattered by a class of concepts C is called the Vapnik-Chervonenkis dimension of the class of concepts C. We will come back to this important definition after learning how to count the threshold functions.

6.3 Regions for two layered networks

We now proceed to formalize the intuitive approach made possible by the graphical representation and examine especially the problem of counting the number of solution regions for perceptron learning defined by a data set.

6.3.1 Regions in weight space for the XOR problem

We can now deal with other aspects of the XOR problem and its solution using a network of three units. Since nine parameters must be defined (two weights and a threshold per unit), weight space is nine-dimensional. We already know that there are sixteen different solutions for the XOR problem with this network, but what is the total number of solution regions for this network?

Let w_1, w_2, w_3 be the weights for the first unit, w_4, w_5, w_6 the weights for the second unit and w_7, w_8, w_9 the weights for the output unit. Let x_1 and x_2 denote the components of the input vector and y_1 and y_2 the outputs of the hidden units. These inputs for each unit define a set of separating hyperplanes in weight space. The set of equations for the two hidden units is

$$\begin{array}{ll} 0 \cdot w_1 + 0 \cdot w_2 + 1 \cdot w_3 = 0 & 0 \cdot w_4 + 0 \cdot w_5 + 1 \cdot w_6 = 0 \\ 0 \cdot w_1 + 1 \cdot w_2 + 1 \cdot w_3 = 0 & 0 \cdot w_4 + 1 \cdot w_5 + 1 \cdot w_6 = 0 \\ 1 \cdot w_1 + 0 \cdot w_2 + 1 \cdot w_3 = 0 & 1 \cdot w_4 + 0 \cdot w_5 + 1 \cdot w_6 = 0 \\ 1 \cdot w_1 + 1 \cdot w_2 + 1 \cdot w_3 = 0 & 1 \cdot w_4 + 1 \cdot w_5 + 1 \cdot w_6 = 0 \end{array}$$

For the output unit there are as many cutting hyperplanes as there are (y_1, y_2)-combinations produced by the hidden units. The form of the equation for each plane is:

$$y_1 w_7 + y_2 w_8 + 1 \cdot w_9 = 0.$$

The separating hyperplanes of the first unit are orthogonal to the separating hyperplanes of the second unit and of the output unit. The first four cuts can generate at most 14 different regions (as in the case of the simple perceptron). The cuts defined by the second unit are also at most 14 and the same happens with the output unit. The maximum total number of regions we get by combining these orthogonal cuts is $14 \cdot 14 \cdot 14$, that is, 2744 polytopes.

This result can be interpreted as meaning that the number of solution regions defined at the surface of the nine-dimensional Boolean sphere is 2744. Sixteen of them are solutions for the XOR problem and 16 for ¬XOR problem. The XOR problem can be solved with this network because the number of solution regions in weight space was increased enormously. Is it possible to proceed as in perceptron learning, by descent on the error function of the network, in order to find an appropriate set of parameters? What is the shape of this error function? Before answering these questions we turn to a topological problem.

6.3.2 Number of regions in general

The capacity of a unit or network depends on the dimension of weight space and the number of cuts with separating hyperplanes. The general question to be answered is: how many regions are defined by m cutting hyperplanes of dimension $n-1$ in an n-dimensional space? We consider only the case of hyperplanes going through the origin but otherwise in *general position*. This means that the intersection of $\ell \leq n$ hyperplanes is of dimension $n - \ell$.

The two-dimensional case is simple: m lines going through the origin define at most $2m$ different regions. Each new line can only go through the cone defined by two previous lines, dividing its two sides in two and adding two new regions in this way.

The three-dimensional case with one, two, or three cuts is simple too. Each cut increases the number of regions by a factor 2. In general: n cuts with $(n-1)$-dimensional hyperplanes in n-dimensional space define 2^n different regions.

The three-dimensional case with four cutting hyperplanes can be solved by projecting on dimension two. The three-dimensional input space is first cut three times with planes in general position. The fourth cutting plane intersects the three previous planes at three different lines. These three lines define a maximum of six regions on the fourth separating hyperplane. This means that the fourth cutting hyperplane divides at most six of the eight existing regions. After the cut with the fourth hyperplane there are six new regions which, added to the eight old ones, gives a total of 14. The general case can be solved using a similar argument.

Proposition 9. *Let $R(m,n)$ denote the number of different regions defined by m separating hyperplanes of dimension $n-1$, in general position, in an n-dimensional space. We set $R(1,n) = 2$ for $n \geq 1$ and $R(m,0) = 0, \forall m \geq 1$. For $n \geq 1$ and $m > 1$*

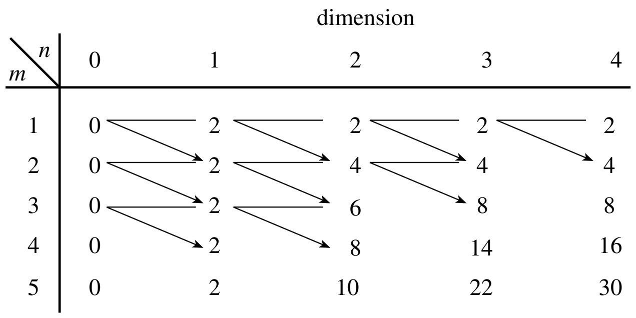
$$R(m,n) = R(m-1,n) + R(m-1,n-1).$$

Proof. The proof is by induction on m. When $m = 2$ and $n = 1$ the formula is valid. If $m = 2$ and $n \geq 2$ we know that $R(2,n) = 4$ and the formula is valid again:

$$R(2,n) = R(1,n) + R(1,n-1) = 2 + 2 = 4.$$

Now $m+1$ hyperplanes of dimension $n-1$ are given in n-dimensional space and in general position ($n \geq 2$). From the induction hypotheses it follows that the first m hyperplanes define $R(m,n)$ regions in n-dimensional space. The hyperplane $m+1$ intersects the first m hyperplanes in m hyperplanes of dimension $n-2$ (since all are in general position). These m hyperplanes divide the $(n-1)$-dimensional space into $R(m,n-1)$ regions. After the cut with the hyperplane $m+1$, exactly$R(m,n-1)$ new regions have been created. The new number of regions is therefore $R(m+1,n) = R(m,n) + R(m,n-1)$ and the proof by induction is complete. □

This result can be represented using a table. Each column of the table corresponds to a different dimension of input space and each row to a different number of separating hyperplanes. The table shows some values for $R(m,n)$.



	dimension				
m \ n	0	1	2	3	4
1	0	2	2	2	2
2	0	2	4	4	4
3	0	2	6	8	8
4	0	2	8	14	16
5	0	2	10	22	30

Fig. 6.20. Recursive calculation of $R(m,n)$

It follows from the table that $R(m,n) = 2^m$ whenever $m \leq n$. This means that the number of regions increases exponentially until the dimension of the space puts a limit to this growth. For $m > n$ the rate of increase becomes polynomial instead of exponential. For $n = 2$ and $n = 3$ we can derive analytic expressions for the polynomials: $R(m,2) = 2m$ and $R(m,3) = m^2 - m + 2$. The following proposition shows that this is not accidental.

Proposition 10. *For $n \geq 1$, $R(m,n)$ is a polynomial of degree $n-1$ on the variable m.*

Proof. The proof follows from induction on n. We denote with $P(a,b)$ a polynomial of degree b on the variable a. The polynomial was explicitly given for $n=2$. For dimension $n+1$ and $m=1$ we know that $R(1,n+1)=2$. If $m>1$ then

$$R(m,n+1) = R(m-1,n+1) + R(m-1,n).$$

Since $R(m-1,n)$ is a polynomial of degree $n-1$ in the variable m it follows that

$$R(m,n+1) = R(m-1,n+1) + P(m,n-1).$$

Repeating this reduction $m-1$ times we finally get

$$\begin{aligned} R(m,n+1) &= R(m-(m-1),n+1) + (m-1)P(m,n-1) \\ &= 2 + (m-1)P(m,n-1) \\ &= P(m,n) \end{aligned}$$

$R(m,n+1)$ is thus a polynomial of degree n in the variable m and the proof by induction is complete. □

A useful formula for $R(m,n) is$

$$R(m,n) = 2\sum_{i=0}^{n-1} \binom{m-1}{i}.$$

The validity of the equation can be proved by induction. It allows us to compute $R(m,n)$ iteratively [271]. Note that this formula tells us how many regions are formed when hyperplanes meet in a general position. In the case of Boolean formulas with Boolean inputs, the hyperplanes in weight space have binary or bipolar coefficients, that is, they do not lie in a general position. The number of regions defined in a 4-dimensional weight space by 8 hyperplanes is, according to the formula, 128. But there are only 104 threshold functions computable with a perceptron with three input lines (and therefore four parameters and eight possible input vectors). The number $R(m,n)$ must then be interpreted as an *upper bound* on the number of logical functions computable with binary inputs.

It is easy to find an upper bound for $R(m,n)$ which can be computed with a few arithmetical operations [457]:

$$R(m,n) < 2\frac{m^n}{n!}.$$

Table 6.2 shows how these bounds behave when the number of inputs n is varied from 1 to 5 [271]. The number of threshold functions of n inputs is denoted in the table by $T(2^n,n)$.

Table 6.2. Comparison of the number of Boolean and threshold functions of n inputs and two different bounds

n	2^{2^n}	$T(2^n, n)$	$R(2^n, n)$	$\lfloor 2^{n^2+1}/n! \rfloor$
1	4	4	4	4
2	16	14	14	16
3	256	104	128	170
4	65,536	1,882	3,882	5,461
5	4.3×10^9	94,572	412,736	559,240

6.3.3 Consequences

Two important consequences become manifest from the analysis just performed.

- *First consequence.* The number of threshold functions in an n-dimensional space grows polynomially whereas the number of possible Boolean functions grows exponentially. The number of Boolean functions definable on n Boolean inputs is 2^{2^n}. The number of threshold functions is a function of the form $2^{n(n-1)}$, since the 2^n input vectors define at most $R(2^n, n)$ regions in weight space, that is, a polynomial of degree $n-1$ on 2^n. The percentage of threshold functions in relation to the total number of logical functions goes to zero as n increases.
- *Second consequence.* In networks with two or more layers we also have learnability problems. Each unit in the first hidden layer separates input space into two halves. If the hidden layer contains m units and the input vector is of dimension n, the maximum number of classification regions is $R(m, n)$. If the number of input vectors is higher, it can happen that not enough classification regions are available to compute a given logical function. Unsolvable problems for all networks with a predetermined number of units can easily be fabricated by increasing the number of input lines into the network.

Let us give an example of a network and its computational limits. The network consists of two units in the hidden layer and one output unit. The extended input vectors are of dimension n. The number of weights in the network is $2n+3$. The number of different input vectors is 2^{n-1}. The number of regions N in weight space defined by the input vectors is bounded by

$$N \leq R(2^{n-1}, n) \cdot R(2^{n-1}, n) \cdot R(4, 3).$$

This means that N is bounded by a function of order $2^{2(n-1)^2}$. Since the number of functions F definable on n inputs is 2^{2^n}, there is a value of n which guarantees $F > N$. Some of the Boolean functions are therefore not computable with this network. Such unsolvable problems can always be found by just overloading the network capacity. The converse is also true: to solve certain problems, the network capacity must be increased if it is not sufficiently high.

6.3.4 The Vapnik–Chervonenkis dimension

Computation of the number of regions defined by m hyperplanes in n-dimensional weight space brings us back to the consideration of the Vapnik-Chervonenkis dimension of a class of concepts and its importance for machine learning.

Assume that a linear separation of the points in the unit square is selected at random by an opponent and we have to find the parameters of the separating line (Figure 6.21). We can select points in the unit square randomly and ask our opponent for their classification, that is, if they belong to the positive or negative half-space. We refine our computation with each new example and we expect to get better and better approximations to the right solution.

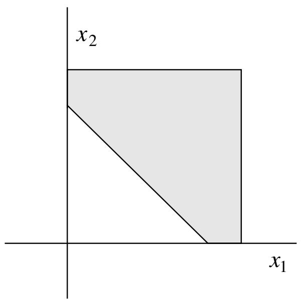


Fig. 6.21. Linear separation of the unit square

Figure 6.22 shows how more examples reduce the range of possible linear separations we can choose from. The linear separations defined by the lines ℓ_1 and ℓ_2 are compatible with the selected points and their classification. All other lines between ℓ_1 and ℓ_2 are also compatible with the known examples. The margin of error, however, is larger in the first than in the second case. Both extreme lines ℓ_1 and ℓ_2 converge asymptotically to the correct linear separation.

If a perceptron is trained with a randomly selected set of examples and tested with another set of points, we call the expected number of correct classifications the *generalization* capability of the perceptron. Generalization becomes better if the training set is larger. In other types of problem in which another class of concepts is used, it is not necessarily so. If, for example, a polynomial of arbitrary degree is fitted to a set of points, it can well happen that the function is *overfitted*, that is, it learns the training set perfectly

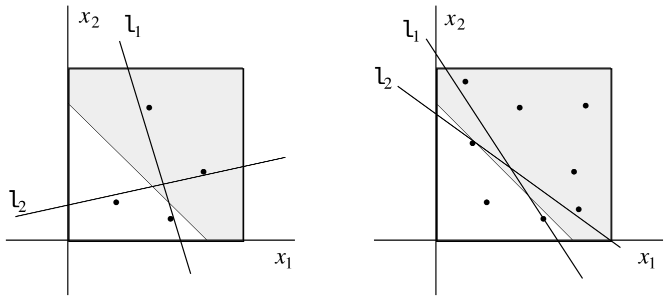


Fig. 6.22. Linear separations compatible with the examples

but interpolates unknown points erroneously. Figure 6.23 shows how this can happen. The learned function oscillates excessively in order to accommodate all points in the training set without error but is very different from the unknown function, which is smoother. More points do not reduce the error as long as our class of concepts consists of all polynomials of arbitrary degree. The only possibility of profiting from more examples is to reduce the size of the search space by, for example, putting a limit on the degree of the acceptable polynomial approximations.

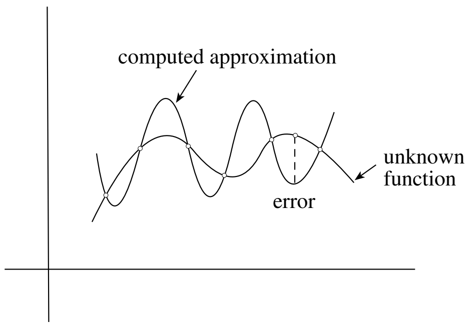


Fig. 6.23. Overfitting a polynomial approximation

A desirable property of the learning process is that it converges to the unknown function with high probability. Additional examples and the minimization of the classification error should bring us monotonically closer to the solution of the problem. We demand that the absolute value of the difference of the approximating function and the unknown function at any point in the input space be less than a given $\varepsilon > 0$.

Neural network's learning consists of approximating an unknown function g with a network function f. Let π_f be the probability that the network function f computes the correct classification of a point chosen randomly in input space. Let ν_f stand for the empirical error rate measured by sampling the input space and testing the classification computed by the network function f. The learning algorithm should guarantee the uniform convergence of ν_f to π_f. In this way, using ν_f as the learning criterion correctly reflects the effective success rate π_f of the network function.

Vapnik and Chervonenkis [437] found a condition for the uniform convergence of ν_f to π_f. They proved the following inequality

$$Pr[\sup_{f\in F} |\nu_f - \pi_f| > \varepsilon] \leq 4\phi(2N)e^{-\varepsilon^2 N/8},$$

which states that the probability that ν_f and π_f of the network function f chosen from the model (the set of computable network functions F) differ by more than ε is smaller than $4\phi(2N)e^{-\varepsilon^2 N/8}$. The variable N stands for the number of examples used and $\phi(2N)$ is the number of binary functions in search space which can be defined over $2N$ examples.

Note that the term $e^{-\varepsilon^2 N/8}$ falls exponentially in the number of examples N. In this case ν_f can come exponentially closer to π_f as long as the number of binary functions definable on $2N$ examples does not grow exponentially. But remember that given a set of points of size $2N$ the number of possible binary labelings is 2^{2N}. The Vapnik-Chervonenkis dimension measures how many binary labelings of a set of points can be computed by one member of a class of concepts. As long as the class of concepts is only capable of covering a polynomial number of these labelings, $\phi(2N)$ will not grow exponentially and will not win the race against the factor $e^{-\varepsilon^2 N/8}$.

Vapnik and Chervonenkis showed that $\phi(2N)$ is bounded by $N^d + 1$, where d is the VC-dimension of the class of concepts. If its VC-dimension is finite we call a class of concepts *learnable*. Perceptron learning is learnable because the VC-dimension of a perceptron with n weights (including the threshold) is finite.

In the case of a perceptron like the one computing the linear separation shown in Figure 6.21, if the empirical success rate $\nu_f = 1$ the probability that it differs from π_f by more than ε is

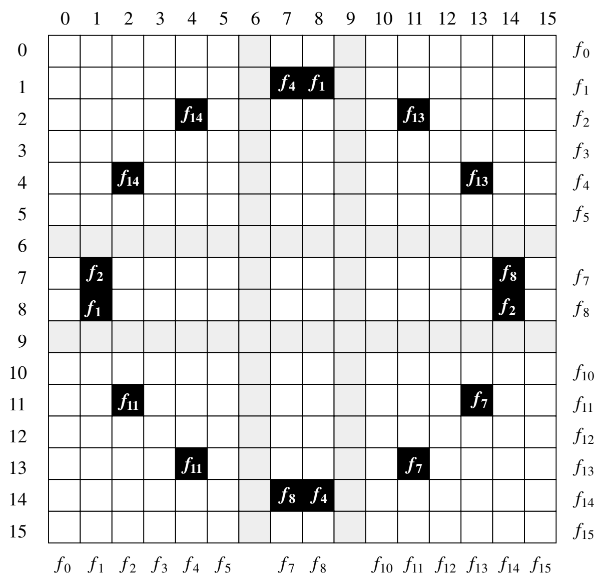
$$Pr[\sup_{f\in F}(1 - \pi_f) > \varepsilon] \leq 4R(2N, n)e^{-\varepsilon^2 N/8}, \tag{6.1}$$

since the number of labelings computable by a perceptron on $2N$ points in general position is equal to $R(2N, n)$, that is, the number of solution regions available in weight space. Since $R(2N, n)$ is a polynomial of degree $n-1$ in the variable N and the term $e^{-\varepsilon^2 N/8}$ goes exponentially to zero, the generalization error margin falls exponentially to zero as the number of examples increases.

6.3.5 The problem of local minima

One of the fundamental problems of iterative learning algorithms is the existence of local minima of the error function. In the case of a single perceptron the error function has a single global minimum region. This is not so with more complex networks.

In the case of the network of three units used in this chapter to compute all solutions for the XOR problem, there are four classes of regions in weight space with associated error from 0 to 4, that is, any subset of the four input vectors can be correctly classified or not. Using an exhaustive search over all possible regions in weight space, it can be shown that there are no spurious local minima in the error function. A path can always be found from regions with error greater than zero to any other region with smaller error.



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
f_4 f_1
f_{14} f_{13}
f_{14} f_{13}
f_2 f_8
f_1 f_2
f_{11} f_7
f_{11} f_7
f_8 f_4
f_0 f_1 f_2 f_3 f_4 f_5 f_7 f_8 f_{10} f_{11} f_{12} f_{13} f_{14} f_{15}
f_0 f_1 f_2 f_3 f_4 f_5 f_7 f_8 f_{10} f_{11} f_{12} f_{13} f_{14} f_{15}

Fig. 6.24. Distribution of the solutions for XOR

Figure 6.24 shows a diagram of the distribution of the solution regions for the XOR problem. Since each of the units in the hidden layer can compute 14 Boolean functions and they are independent, and since the associated cuts in weight space are orthogonal, we can divide the surface of the nine-dimensional Boolean sphere into 14×14 regions. Each one of these is subdivided by the 14 regions defined by the output unit. The column labeling of the diagram corresponds to the function computed by the first hidden unit, the row labeling to the function computed by the second hidden unit. The dark regions indicate where a solution region for the XOR problem can be found and which function is computed by the output unit. The shadowed columns and rows correspond to the XOR and $\neg$XOR functions, which are not computable by each of the hidden units. The diagram shows that the solution regions are distributed symmetrically on the surface of the Boolean sphere. Although this simple diagram cannot capture the whole complexity of the neighborhood relations in a nine-dimensional space, it gives a pretty good idea of the actual distribution of solutions. The symmetrical distribution of the solution regions is important, because we can start randomly from any point in weight space and a solution is never very far away. The problem, however, is deciding in what direction to start the search. We will deal with this problem in the next chapter by generalizing the kind of activation functions acceptable in each unit.

6.4 Historical and bibliographical remarks

Although networks with several layers of computing units were proposed right at the beginning of the development of neural network models, the problem which limited their applicability was that no reliable learning method was known. Rosenblatt experimented with a kind of learning in which the error at the output was propagated to elements in the first layers of computing units.

Another important problem is the location and number of local minima of the error function which can lead the learning algorithm astray. Hecht-Nielsen [186] and Poston et al. [349] have discussed the structure of the error function. Others, like Hush et al. [206], developed similar visualization methods to explore the shape of the error function.

Threshold functions were studied intensively in the 1960s and the bounds on the number of threshold functions given in this chapter were derived at that time. It is possible to characterize any threshold function of n inputs uniquely by a set of $n+1$ parameters, as was shown by Chow and by Dertouzos [401]. The Chow coefficients correspond to the centroid of the vertices with function value 1 on the n-dimensional binary hypercube.

Much research has been done on the topological properties of polytopes and figures on spheres [388]. Nilsson [329] and others studied the importance of the number of regions defined by cutting hyperplanes relatively early. The number of regions defined by cuts in an n-dimensional space was studied in more general form by Euler [29]. The relation between learnability and the Euler characteristic was studied by Minsky and Papert [312].

In the 1970s it became clear that Vapnik and Chervonenkis' approach provided the necessary tools for a general definition of "learnable problems". Valiant [436] was one of the first to propose such a model-independent theory of learning. In this approach the question to be answered is whether the search space in the domain of learnable functions can be restricted in polynomial time. The VC-dimension of the class of concepts can thus help to determine learnability. The VC-dimension of some network architectures has been studied by Baum [44]. Some authors have studied the VC-dimension of other interesting classes of concepts. For example, the VC-dimension of sparse polynomials over the reals, that is polynomials with at most t monomials, is linear in t and thus this class of concepts can be uniformly learned [238]. Such sparse polynomials have a finite VC-dimension because they do not have enough plasticity to shatter an infinite number of points, since their number of different roots is bounded by $2t - 1$.

Exercises

1. Consider the Boolean functions of two arguments. Write a computer program to measure the relative sizes of the 14 solution regions for perceptron learning.
2. Figure 6.19 shows that eight points on the plane can produce non-learnable concepts for two linear separations. What is the *minimum* number of points in $\mathbb{R}^2$ which can produce a non-learnable concept using two linear separations?
3. Write a computer program to test the validity of equation (6.1) for a linear separation of the type shown in Figure 6.21.
4. Consider a perceptron that accepts complex inputs x_1, x_2. The weights w_1, w_2 are also complex numbers, and the threshold is zero. The perceptron fires if the condition $\text{Re}(x_1 w_1 + x_2 w_2) \geq \text{Im}(x_1 w_1 + x_2 w_2)$ is satisfied. The binary input 0 is coded as the complex number $(1, 0)$ and the binary input 1 as the number $(0, 1)$. How many of the logical functions of two binary arguments can be computed with this system? Can XOR be computed?
5. Construct a non-learnable concept in $\mathbb{R}^2$ for three linear separations.

7

The Backpropagation Algorithm

7.1 Learning as gradient descent

We saw in the last chapter that multilayered networks are capable of computing a wider range of Boolean functions than networks with a single layer of computing units. However the computational effort needed for finding the correct combination of weights increases substantially when more parameters and more complicated topologies are considered. In this chapter we discuss a popular learning method capable of handling such large learning problems — *the backpropagation algorithm.* This numerical method was used by different research communities in different contexts, was discovered and rediscovered, until in 1985 it found its way into connectionist AI mainly through the work of the PDP group [382]. It has been one of the most studied and used algorithms for neural networks learning ever since.

In this chapter we present a proof of the backpropagation algorithm based on a graphical approach in which the algorithm reduces to a graph labeling problem. This method is not only more general than the usual analytical derivations, which handle only the case of special network topologies, but also much easier to follow. It also shows how the algorithm can be efficiently implemented in computing systems in which only local information can be transported through the network.

7.1.1 Differentiable activation functions

The backpropagation algorithm looks for the minimum of the error function in weight space using the method of gradient descent. The combination of weights which minimizes the error function is considered to be a solution of the learning problem. Since this method requires computation of the gradient of the error function at each iteration step, we must guarantee the continuity and differentiability of the error function. Obviously we have to use a kind of activation function other than the step function used in perceptrons, because the composite function produced by interconnected perceptrons is discontinuous, and therefore the error function too. One of the more popular activation functions for backpropagation networks is the *sigmoid*, a real function $s_c : \mathbb{R} \rightarrow (0,1)$ defined by the expression

$$s_c(x) = \frac{1}{1 + \mathrm{e}^{-cx}}.$$

The constant c can be selected arbitrarily and its reciprocal $1/c$ is called the temperature parameter in stochastic neural networks. The shape of the sigmoid changes according to the value of c, as can be seen in Figure 7.1. The graph shows the shape of the sigmoid for $c = 1$, $c = 2$ and $c = 3$. Higher values of c bring the shape of the sigmoid closer to that of the step function and in the limit $c \rightarrow \infty$ the sigmoid converges to a step function at the origin. In order to simplify all expressions derived in this chapter we set $c = 1$, but after going through this material the reader should be able to generalize all the expressions for a variable c. In the following we call the sigmoid $s_1(x)$ just $s(x)$.

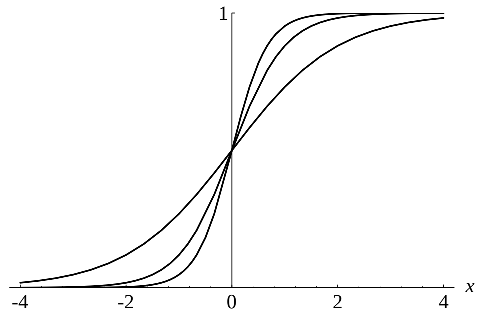


Fig. 7.1. Three sigmoids (for $c = 1$, $c = 2$ and $c = 3$)

The derivative of the sigmoid with respect to x, needed later on in this chapter, is

$$\frac{d}{dx}s(x) = \frac{\mathrm{e}^{-x}}{(1 + \mathrm{e}^{-x})^2} = s(x)(1 - s(x)).$$

We have already shown that, in the case of perceptrons, a symmetrical activation function has some advantages for learning. An alternative to the sigmoid is the symmetrical sigmoid $S(x)$ defined as

$$S(x) = 2s(x) - 1 = \frac{1 - \mathrm{e}^{-x}}{1 + \mathrm{e}^{-x}}.$$

This is nothing but the hyperbolic tangent for the argument $x/2$ whose shape is shown in Figure 7.2 (upper right). The figure shows four types of continuous “squashing” functions. The ramp function (lower right) can also be used in learning algorithms taking care to avoid the two points where the derivative is undefined.

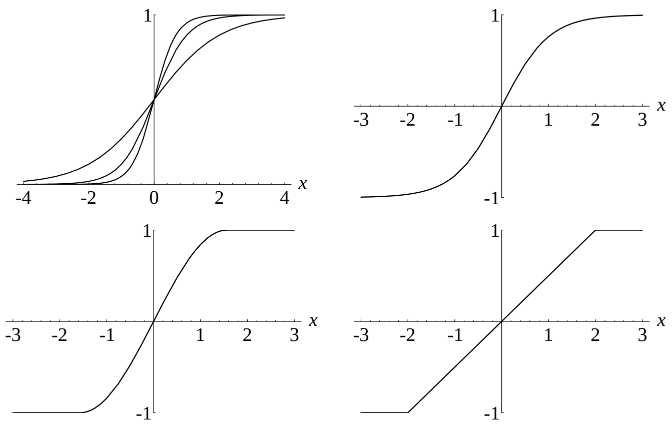


Fig. 7.2. Graphics of some "squashing" functions

Many other kinds of activation functions have been proposed and the backpropagation algorithm is applicable to all of them. A differentiable activation function makes the function computed by a neural network differentiable (assuming that the integration function at each node is just the sum of the inputs), since the network itself computes only function compositions. The error function also becomes differentiable.

Figure 7.3 shows the smoothing produced by a sigmoid in a step of the error function. Since we want to follow the gradient direction to find the minimum of this function, it is important that no regions exist in which the error function is completely flat. As the sigmoid always has a positive derivative, the slope of the error function provides a greater or lesser descent direction which can be followed. We can think of our search algorithm as a physical process in which a small sphere is allowed to roll on the surface of the error function until it reaches the bottom.

7.1.2 Regions in input space

The sigmoid's output range contains all numbers strictly between 0 and 1. Both extreme values can only be reached asymptotically. The computing units considered in this chapter evaluate the sigmoid using the net amount of excitation as its argument. Given weights $w_1, \dots, w_n$ and a bias $-\theta$, a sigmoidal unit computes for the input $x_1, \dots, x_n$ the output

$$\frac{1}{1 + \exp\left(\sum_{i=1}^{n} w_i x_i - \theta\right)}.$$

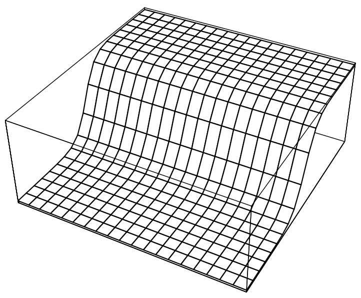


Fig. 7.3. A step of the error function

A higher net amount of excitation brings the unit's output nearer to 1. The continuum of output values can be compared to a division of the input space in a continuum of classes. A higher value of c makes the separation in input space sharper.

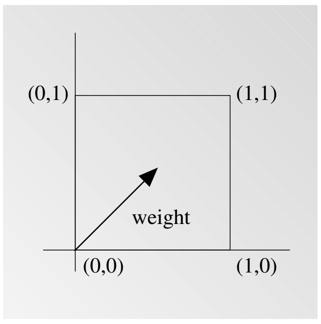


Fig. 7.4. Continuum of classes in input space

Note that the step of the sigmoid is normal to the vector $(w_1, \ldots, w_n, -\theta)$ so that the weight vector points in the direction in extended input space in which the output of the sigmoid changes faster.

7.1.3 Local minima of the error function

A price has to be paid for all the positive features of the sigmoid as activation function. The most important problem is that, under some circumstances, local minima appear in the error function which would not be there if the step function had been used. Figure 7.5 shows an example of a local minimum with a higher error level than in other regions. The function was computed for a single unit with two weights, constant threshold, and four input-output patterns in the training set. There is a valley in the error function and if gradient descent is started there the algorithm will not converge to the global minimum.

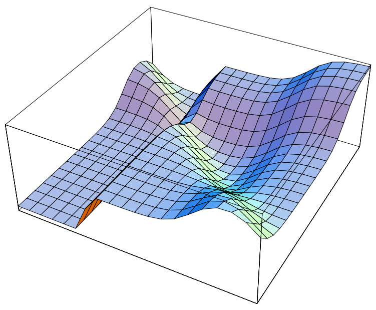


Fig. 7.5. A local minimum of the error function

In many cases local minima appear because the targets for the outputs of the computing units are values other than 0 or 1. If a network for the computation of XOR is trained to produce 0.9 at the inputs (0,1) and (1,0) then the surface of the error function develops some protuberances, where local minima can arise. In the case of binary target values some local minima are also present, as shown by Lisboa and Perantonis who analytically found all local minima of the XOR function [277].

7.2 General feed-forward networks

In this section we show that backpropagation can easily be derived by linking the calculation of the gradient to a graph labeling problem. This approach is not only elegant, but also more general than the traditional derivations found in most textbooks. General network topologies are handled right from the beginning, so that the proof of the algorithm is not reduced to the multilayered case. Thus one can have it both ways, more general yet simpler [375].

7.2.1 The learning problem

Recall that in our general definition a feed-forward neural network is a computational graph whose nodes are computing units and whose directed edges transmit numerical information from node to node. Each computing unit is capable of evaluating a single primitive function of its input. In fact the network represents a chain of function compositions which transform an input to an output vector (called a pattern). The network is a particular implementation of a composite function from input to output space, which we call the *network function*. The learning problem consists of finding the optimal combination of weights so that the network function φ approximates a given function f as closely as possible. However, we are not given the function f *explicitly* but only implicitly through some examples.

Consider a feed-forward network with n input and m output units. It can consist of any number of hidden units and can exhibit any desired feed-forward connection pattern. We are also given a training set $\{(\mathbf{x}_1, \mathbf{t}_1), \ldots, (\mathbf{x}_p, \mathbf{t}_p)\}$ consisting of p ordered pairs of n- and m-dimensional vectors, which are called the input and output patterns. Let the primitive functions at each node of the network be continuous and differentiable. The weights of the edges are real numbers selected at random. When the input pattern $\mathbf{x}_i$ from the training set is presented to this network, it produces an output $\mathbf{o}_i$ different in general from the target $\mathbf{t}_i$. What we want is to make $\mathbf{o}_i$ and $\mathbf{t}_i$ identical for $i = 1, \ldots, p$, by using a learning algorithm. More precisely, we want to minimize the error function of the network, defined as

$$E = \frac{1}{2} \sum_{i=1}^{p} \|\mathbf{o}_i - \mathbf{t}_i\|^2.$$

After minimizing this function for the training set, new unknown input patterns are presented to the network and we expect it to *interpolate*. The network must recognize whether a new input vector is similar to learned patterns and produce a similar output.

The backpropagation algorithm is used to find a local minimum of the error function. The network is initialized with randomly chosen weights. The gradient of the error function is computed and used to correct the initial weights. Our task is to compute this gradient recursively.

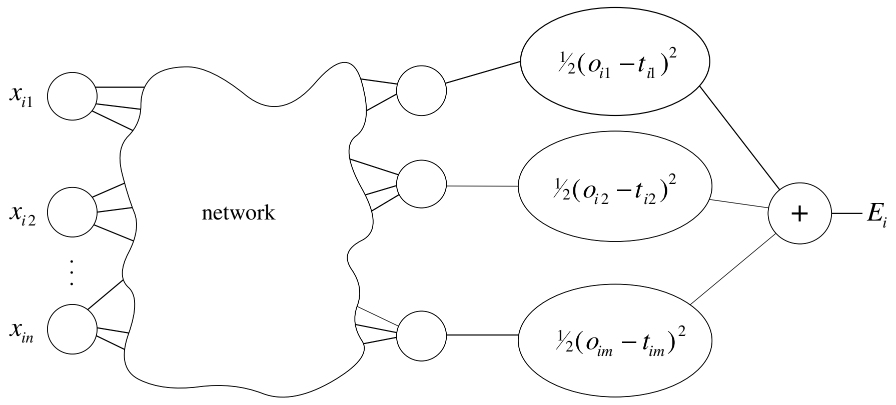

x_{i1}
x_{i2}
x_{in}
network
$\frac{1}{2}(o_{i1} - t_{i1})^2$
$\frac{1}{2}(o_{i2} - t_{i2})^2$
$\frac{1}{2}(o_{im} - t_{im})^2$
+
E_i

Fig. 7.6. Extended network for the computation of the error function

The first step of the minimization process consists of extending the network, so that it computes the error function automatically. Figure 7.6 shows how this is done. Every one of the j output units of the network is connected to a node which evaluates the function $\frac{1}{2}(o_{ij} - t_{ij})^2$, where o_{ij} and t_{ij} denote the j-th component of the output vector $\mathbf{o}_i$ and of the target $\mathbf{t}_i$. The outputs of the additional m nodes are collected at a node which adds them up and gives the sum E_i as its output. The same network extension has to be built for each pattern $\mathbf{t}_i$. A computing unit collects all quadratic errors and outputs their sum $E_1 + \cdots + E_p$. The output of this extended network is the error function E.

We now have a network capable of calculating the total error for a given training set. The weights in the network are the only parameters that can be modified to make the quadratic error E as low as possible. Because E is calculated by the extended network exclusively through composition of the node functions, it is a continuous and differentiable function of the ℓ weights $w_1, w_2, \ldots, w_\ell$ in the network. We can thus minimize E by using an iterative process of gradient descent, for which we need to calculate the gradient

$$\nabla E = \left(\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_\ell}\right).$$

Each weight is updated using the increment

$$\Delta w_i = -\gamma \frac{\partial E}{\partial w_i} \quad \text{for } i = 1, \ldots, \ell,$$

where γ represents a learning constant, i.e., a proportionality parameter which defines the step length of each iteration in the negative gradient direction.

With this extension of the original network the whole learning problem now reduces to the question of calculating the gradient of a network function with respect to its weights. Once we have a method to compute this gradient, we can adjust the network weights iteratively. In this way we expect to find a minimum of the error function, where $\nabla E = 0$.

7.2.2 Derivatives of network functions

Now forget everything about training sets and learning. Our objective is to find a method for efficiently calculating the gradient of a one-dimensional network function according to the weights of the network. Because the network is equivalent to a complex chain of function compositions, we expect the chain rule of differential calculus to play a major role in finding the gradient of the function. We take account of this fact by giving the nodes of the network a composite structure. Each node now consists of a left and a right side, as shown in Figure 7.7. We call this kind of representation a *B-diagram* (for backpropagation diagram).

The right side computes the primitive function associated with the node, whereas the left side computes the derivative of this primitive function for the same input.

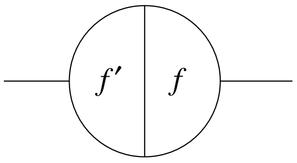


Fig. 7.7. The two sides of a computing unit

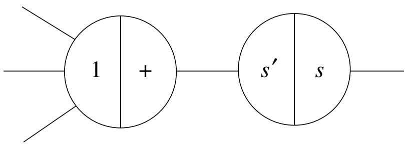


Fig. 7.8. Separation of integration and activation function

Note that the integration function can be separated from the activation function by splitting each node into two parts, as shown in Figure 7.8. The first node computes the sum of the incoming inputs, the second one the activation function s. The derivative of s is s' and the partial derivative of the sum of n arguments with respect to any one of them is just 1. This separation simplifies our discussion, as we only have to think of a single function which is being computed at each node and not of two.

The network is evaluated in two stages: in the first one, the feed-forward step, information comes from the left and each unit evaluates its primitive function f in its right side as well as the derivative f' in its left side. Both results are stored in the unit, but only the result from the right side is transmitted to the units connected to the right. The second step, the backpropagation step, consists in running the whole network backwards, whereby the stored results are now used. There are three main cases which we have to consider.

First case: function composition

The B-diagram of Figure 7.9 contains only two nodes. In the feed-forward step, incoming information into a unit is used as the argument for the evaluation of the node's primitive function and its derivative. In this step the network computes the composition of the functions f and g. Figure 7.10 shows the state of the network after the feed-forward step. The correct result of the function composition has been produced at the output unit and each unit has stored some information on its left side.

In the backpropagation step the input from the right of the network is the constant 1. Incoming information to a node is *multiplied* by the value stored in its left side. The result of the multiplication is transmitted to the next unit to the left. We call the result at each node the *traversing value* at this node. Figure 7.11 shows the final result of the backpropagation step, which is $f'(g(x))g'(x)$, i.e., the derivative of the function composition $f(g(x))$

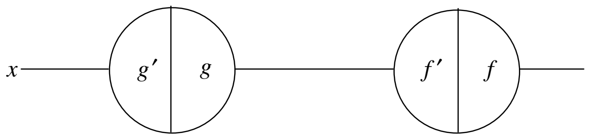


Fig. 7.9. Network for the composition of two functions

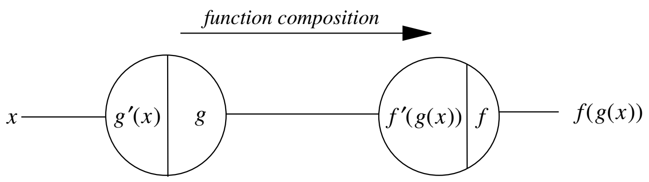


Fig. 7.10. Result of the feed-forward step

implemented by this network. The backpropagation step provides an implementation of the chain rule. Any sequence of function compositions can be evaluated in this way and its derivative can be obtained in the backpropagation step. We can think of the network as being used backwards with the input 1, whereby at each node the product with the value stored in the left side is computed.

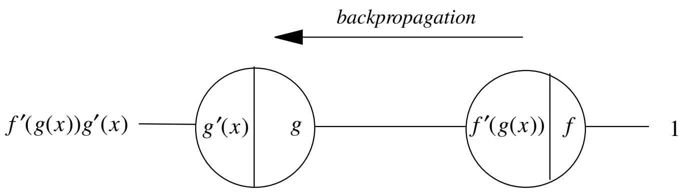


Fig. 7.11. Result of the backpropagation step

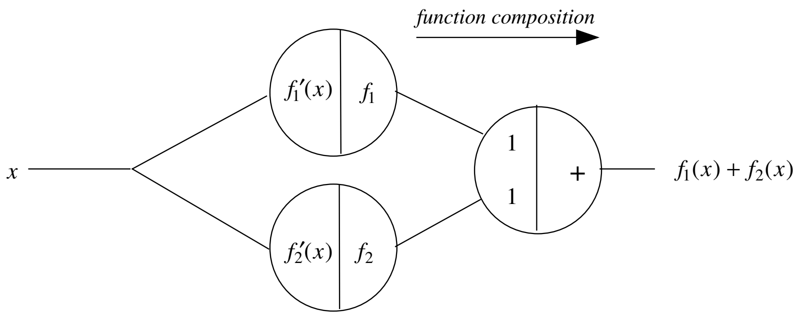


Fig. 7.12. Addition of functions

Second case: function addition

The next case to consider is the addition of two primitive functions. Figure 7.12 shows a network for the computation of the addition of the functions f_1 and f_2 . The additional node has been included to handle the addition of the two functions. The partial derivative of the addition function with respect to any one of the two inputs is 1. In the feed-forward step the network computes the result $f_1(x) + f_2(x)$. In the backpropagation step the constant 1 is fed from the left side into the network. All incoming edges to a unit fan out the traversing value at this node and distribute it to the connected units to the left. Where two right-to-left paths meet, the computed traversing values are added. Figure 7.13 shows the result $f_1'(x) + f_2'(x)$ of the backpropagation step, which is the derivative of the function addition $f_1 + f_2$ evaluated at x. A simple proof by induction shows that the derivative of the addition of any number of functions can be handled in the same way.

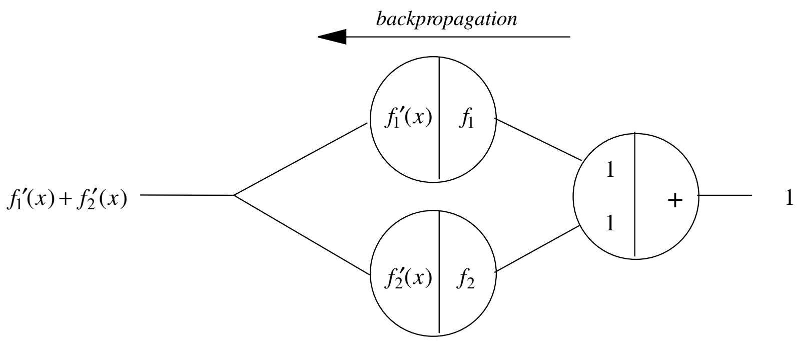


Fig. 7.13. Result of the backpropagation step

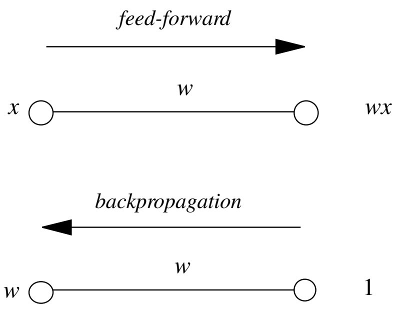


Fig. 7.14. Forward computation and backpropagation at an edge

Third case: weighted edges

Weighted edges could be handled in the same manner as function compositions, but there is an easier way to deal with them. In the feed-forward step the incoming information x is multiplied by the edge's weight w. The result is wx. In the backpropagation step the traversing value 1 is multiplied by the weight of the edge. The result is w, which is the derivative of wx with respect to x. From this we conclude that weighted edges are used in exactly the same way in both steps: they modulate the information transmitted in each direction by multiplying it by the edges' weight.

7.2.3 Steps of the backpropagation algorithm

We can now formulate the complete backpropagation algorithm and prove by induction that it works in arbitrary feed-forward networks with differentiable activation functions at the nodes. We assume that we are dealing with a network with a single input and a single output unit.

Algorithm 7.2.1 *Backpropagation algorithm.*

Consider a network with a single real input x and network function F. The derivative $F'(x)$ is computed in two phases:

Feed-forward: the input x is fed into the network. The primitive functions at the nodes and their derivatives are evaluated at each node. The derivatives are stored.

Backpropagation: the constant 1 is fed into the output unit and the network is run backwards. Incoming information to a node is added and the result is multiplied by the value stored in the left part of the unit. The result is transmitted to the left of the unit. The result collected at the input unit is the derivative of the network function with respect to x.

The following proposition shows that the algorithm is correct.

Proposition 11. *Algorithm 7.2.1 computes the derivative of the network function F with respect to the input x correctly.*

Proof. We have already shown that the algorithm works for units in series, units in parallel and also for weighted edges. Let us make the induction assumption that the algorithm works for any feed-forward network with n or fewer nodes. Consider now the B-diagram of Figure 7.15, which contains $n+1$ nodes. The feed-forward step is executed first and the result of the single output unit is the network function F evaluated at x. Assume that m units, whose respective outputs are $F_1(x), \ldots, F_m(x)$ are connected to the output unit. Since the primitive function of the output unit is φ, we know that

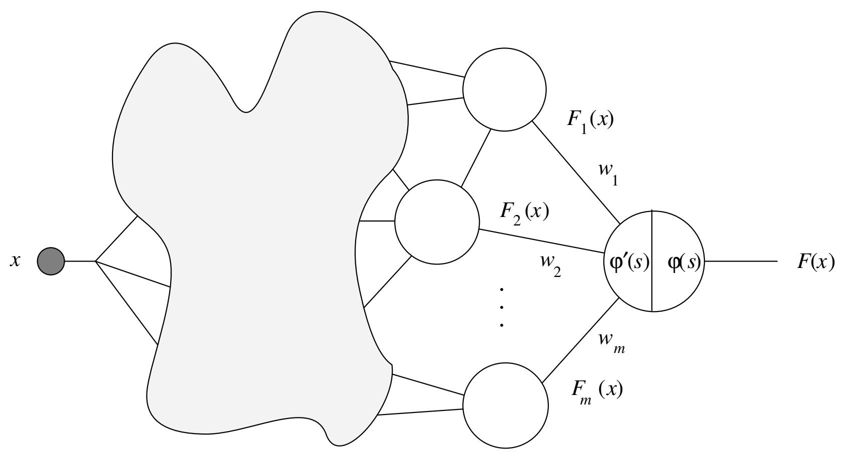


Fig. 7.15. Backpropagation at the last node

$$F(x) = \varphi(w_1F_1(x) + w_2F_2(x) + \cdots + w_mF_m(x)).$$

The derivative of F at x is thus

$$F'(x) = \varphi'(s)(w_1F_1'(x) + w_2F_2'(x) + \cdots + w_mF_m'(x)),$$

where $s = \varphi(w_1F_1(x) + w_2F_2(x) + \cdots + w_mF_m(x))$. The subgraph of the main graph which includes all possible paths from the input unit to the unit whose output is $F_1(x)$ defines a subnetwork whose network function is F_1 and which consists of n or fewer units. By the induction assumption we can calculate the derivative of F_1 at x, by introducing a 1 into the unit and running the subnetwork backwards. The same can be done with the units whose outputs are $F_2(x), \ldots, F_m(x)$. If instead of 1 we introduce the constant $\varphi'(s)$ and multiply it by w_1 we get $w_1F_1'(x)\varphi'(s)$ at the input unit in the backpropagation step. Similarly we get $w_2F_2'(x)\varphi'(s), \ldots, w_mF_m'(x)\varphi'(s)$ for the rest of the units. In the backpropagation step with the whole network we add these m results and we finally get

$$\varphi'(s)(w_1F_1'(x) + w_2F_2'(x) + \cdots + w_mF_m'(x))$$

which is the derivative of F evaluated at x. Note that introducing the constants $w_1\varphi'(s), \ldots, w_m\varphi'(s)$ into the m units connected to the output unit can be done by introducing a 1 into the output unit, multiplying by the stored value $\varphi'(s)$ and distributing the result to the m units through the edges with weights $w_1, w_2, \ldots, w_m$. We are in fact running the network backwards as the backpropagation algorithm demands. This means that the algorithm works with networks of $n+1$ nodes and this concludes the proof. □

Implicit in the above analysis is that all inputs to a node are added before the one-dimensional activation function is computed. We can consider also activation functions f of several variables, but in this case the left side of the unit stores all partial derivatives of f with respect to each variable. Figure 7.16 shows an example for a function f of two variables x_1 and x_2, delivered through two different edges. In the backpropagation step each stored partial derivative is multiplied by the traversing value at the node and transmitted to the left *through its own edge.* It is easy to see that backpropagation still works in this more general case.

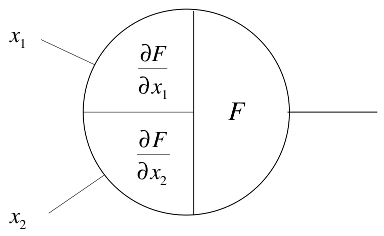

x_1
$\frac{\partial F}{\partial x_1}$
F
$\frac{\partial F}{\partial x_2}$
x_2

Fig. 7.16. Stored partial derivatives at a node

The backpropagation algorithm also works correctly for networks with more than one input unit in which several independent variables are involved. In a network with two inputs for example, where the independent variables x_1 and x_2 are fed into the network, the network result can be called $F(x_1, x_2)$. The network function now has two arguments and we can compute the partial derivative of F with respect to x_1 or x_2. The feed-forward step remains unchanged and all left side slots of the units are filled as usual. However, in the backpropagation step we can identify two subnetworks: one consists of all paths connecting the first input unit to the output unit and another of all paths from the second input unit to the output unit. By applying the backpropagation step in the first subnetwork we get the partial derivative of F with respect to x_1 at the first input unit. The backpropagation step on the second subnetwork yields the partial derivative of F with respect to x_2 at the second input unit. Note that we can overlap both computations and perform a single backpropagation step over the whole network. We still get the same results.

7.2.4 Learning with backpropagation

We consider again the learning problem for neural networks. Since we want to minimize the error function E, which depends on the network weights, we have to deal with all weights in the network one at a time. The feed-forward step is computed in the usual way, but now we also store the output of each unit in its right side. We perform the backpropagation step in the extended network that computes the error function and we then fix our attention on one of the weights, say w_{ij} whose associated edge points from the i-th to the

j-th node in the network. This weight can be treated as an input channel into the subnetwork made of all paths starting at w_{ij} and ending in the single output unit of the network. The information fed into the subnetwork in the feed-forward step was $o_i w_{ij}$, where o_i is the stored output of unit i. The backpropagation step computes the gradient of E with respect to this input, i.e., $\partial E/\partial o_i w_{ij}$. Since in the backpropagation step o_i is treated as a constant, we finally have

$$\frac{\partial E}{\partial w_{ij}} = o_i \frac{\partial E}{\partial o_i w_{ij}}.$$

Summarizing, the backpropagation step is performed in the usual way. All subnetworks defined by each weight of the network can be handled simultaneously, but we now store additionally at each node i:

- The output o_i of the node in the feed-forward step.
- The cumulative result of the backward computation in the backpropagation step up to this node. We call this quantity the *backpropagated error.*

If we denote the backpropagated error at the j-th node by δ_j, we can then express the partial derivative of E with respect to w_{ij} as:

$$\frac{\partial E}{\partial w_{ij}} = o_i \delta_j.$$

Once all partial derivatives have been computed, we can perform gradient descent by adding to each weight w_{ij} the increment

$$\Delta w_{ij} = -\gamma o_i \delta_j.$$

This correction step is needed to transform the backpropagation algorithm into a learning method for neural networks.

This graphical proof of the backpropagation algorithm applies to arbitrary feed-forward topologies. The graphical approach also immediately suggests hardware implementation techniques for backpropagation.

7.3 The case of layered networks

An important special case of feed-forward networks is that of layered networks with one or more hidden layers. In this section we give explicit formulas for the weight updates and show how they can be calculated using linear algebraic operations. We also show how to label each node with the backpropagated error in order to avoid redundant computations.

7.3.1 Extended network

We will consider a network with n input sites, k hidden, and m output units. The weight between input site i and hidden unit j will be called $w_{ij}^{(1)}$. The weight between hidden unit i and output unit j will be called $w_{ij}^{(2)}$. The bias $-\theta$ of each unit is implemented as the weight of an additional edge. Input vectors are thus extended with a 1 component, and the same is done with the output vector from the hidden layer. Figure 7.17 shows how this is done. The weight between the constant 1 and the hidden unit j is called $w_{n+1,j}^{(1)}$ and the weight between the constant 1 and the output unit j is denoted by $w_{k+1,j}^{(2)}$.

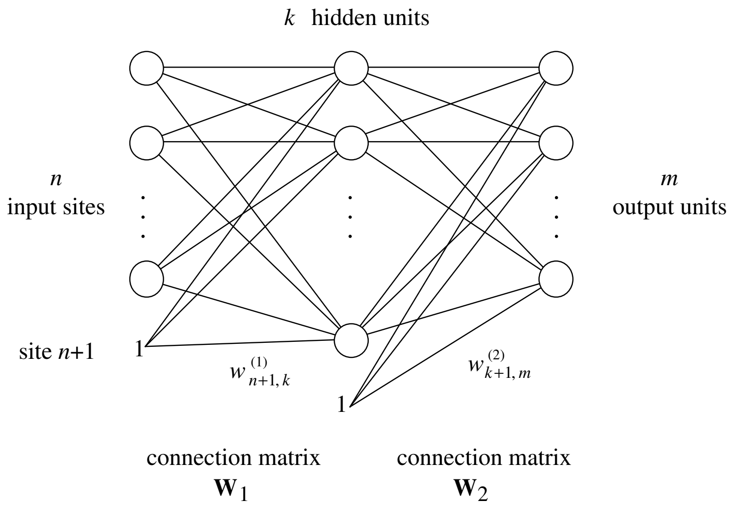


Fig. 7.17. Notation for the three-layered network

There are $(n+1) \times k$ weights between input sites and hidden units and $(k+1) \times m$ between hidden and output units. Let $\overline{\mathbf{W}}_1$ denote the $(n+1) \times k$ matrix with component $w_{ij}^{(1)}$ at the i-th row and the j-th column. Similarly let $\overline{\mathbf{W}}_2$ denote the $(k+1) \times m$ matrix with components $w_{ij}^{(2)}$. We use an overlined notation to emphasize that the last row of both matrices corresponds to the biases of the computing units. The matrix of weights without this last row will be needed in the backpropagation step. The n-dimensional input vector $\mathbf{o} = (o_1, \ldots, o_n)$ is extended, transforming it to $\hat{\mathbf{o}} = (o_1, \ldots, o_n, 1)$. The excitation net_j of the j-th hidden unit is given by

$$net_j = \sum_{i=1}^{n+1} w_{ij}^{(1)} \hat{o}_i .$$

The activation function is a sigmoid and the output $o_j^{(1)}$ of this unit is thus

$$o_j^{(1)} = s\left(\sum_{i=1}^{n+1} w_{ij}^{(1)} \hat{o}_i\right).$$

The excitation of all units in the hidden layer can be computed with the vector-matrix multiplication $\hat{\mathbf{o}}\overline{\mathbf{W}}_1$. The vector $\mathbf{o}^{(1)}$ whose components are the outputs of the hidden units is given by

$$\mathbf{o}^{(1)} = s(\hat{\mathbf{o}}\overline{\mathbf{W}}_1),$$

using the convention of applying the sigmoid to each component of the argument vector. The excitation of the units in the output layer is computed using the extended vector $\hat{\mathbf{o}}^{(1)} = (o_1^{(1)}, \ldots, o_k^{(1)}, 1)$. The output of the network is the m-dimensional vector $\mathbf{o}^{(2)}$, where

$$\mathbf{o}^{(2)} = s(\hat{\mathbf{o}}^{(1)}\overline{\mathbf{W}}_2).$$

These formulas can be generalized for any number of layers and allow direct computation of the flow of information in the network with simple matrix operations.

7.3.2 Steps of the algorithm

Figure 7.18 shows the extended network for computation of the error function. In order to simplify the discussion we deal with a single input-output pair $(\mathbf{o}, \mathbf{t})$ and generalize later to p training examples. The network has been extended with an additional layer of units. The right sides compute the quadratic deviation $\frac{1}{2}(o_i^{(2)} - t_i)$ for the i-th component of the output vector and the left sides store $(o_i^{(2)} - t_i)$. Each output unit i in the original network computes the sigmoid s and produces the output $o_i^{(2)}$. Addition of the quadratic deviations gives the error E. The error function for p input-output examples can be computed by creating p networks like the one shown, one for each training pair, and adding the outputs of all of them to produce the total error of the training set.

After choosing the weights of the network randomly, the backpropagation algorithm is used to compute the necessary corrections. The algorithm can be decomposed in the following four steps:

i) Feed-forward computation
ii) Backpropagation to the output layer
iii) Backpropagation to the hidden layer
iv) Weight updates

The algorithm is stopped when the value of the error function has become sufficiently small.

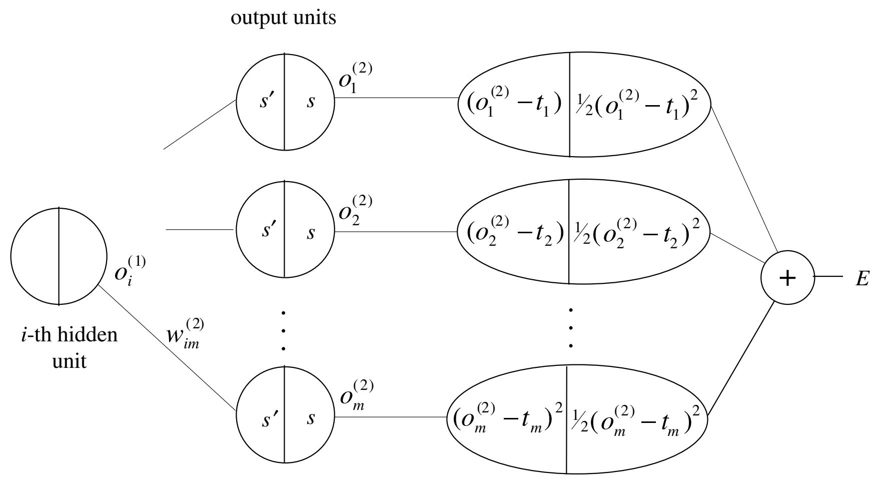


Fig. 7.18. Extended multilayer network for the computation of E

First step: feed-forward computation

The vector **o** is presented to the network. The vectors $\mathbf{o}^{(1)}$ and $\mathbf{o}^{(2)}$ are computed and stored. The evaluated derivatives of the activation functions are also stored at each unit.

Second step: backpropagation to the output layer

We are looking for the first set of partial derivatives $\partial E/\partial w_{ij}^{(2)}$. The backpropagation path from the output of the network up to the output unit j is shown in the B-diagram of Figure 7.19.

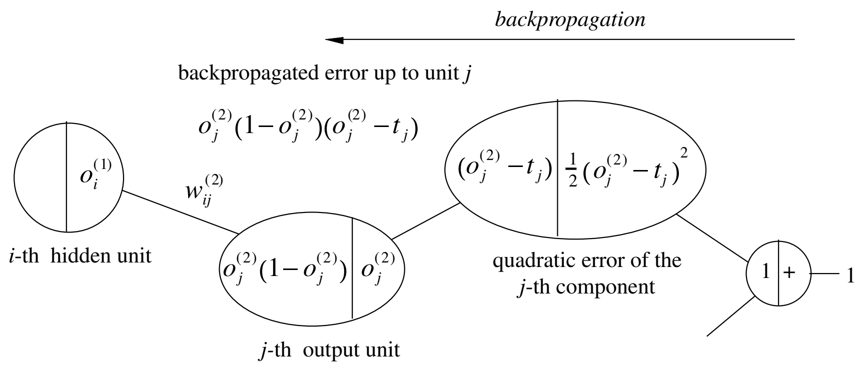

backpropagation
backpropagated error up to unit j
$o_j^{(2)}(1-o_j^{(2)})(o_j^{(2)}-t_j)$
$o_i^{(1)}$
$w_{ij}^{(2)}$
i-th hidden unit
$(o_j^{(2)}-t_j)$
$\frac{1}{2}(o_j^{(2)}-t_j)^2$
quadratic error of the j-th component
$o_j^{(2)}(1-o_j^{(2)})$
$o_j^{(2)}$
j-th output unit
1
+
1

Fig. 7.19. Backpropagation path up to output unit j

From this path we can collect by simple inspection all the multiplicative terms which define the backpropagated error $\delta_j^{(2)}$. Therefore

$$\delta_j^{(2)} = o_j^{(2)}(1 - o_j^{(2)})(o_j^{(2)} - t_j),$$

and the partial derivative we are looking for is

$$\frac{\partial E}{\partial w_{ij}^{(2)}} = [o_j^{(2)}(1 - o_j^{(2)})(o_j^{(2)} - t_j)]o_i^{(1)} = \delta_j^{(2)} o_i^{(1)}.$$

Remember that for this last step we consider the weight $w_{ij}^{(2)}$ to be a variable and its input $o_i(1)$ a constant.

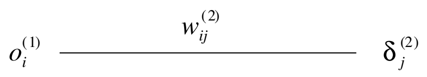


Fig. 7.20. Input and backpropagated error at an edge

Figure 7.20 shows the general situation we find during the backpropagation algorithm. At the input side of the edge with weight w_{ij} we have $o_i^{(1)}$ and at the output side the backpropagated error $\delta_j^{(2)}$.

Third step: backpropagation to the hidden layer

Now we want to compute the partial derivatives $\partial E / \partial w_{ij}^{(1)}$. Each unit j in the hidden layer is connected to each unit q in the output layer with an edge of weight $w_{jq}^{(2)}$, for $q = 1, \ldots, m$. The backpropagated error up to unit j in the hidden layer must be computed taking into account all possible backward paths, as shown in Figure 7.21. The backpropagated error is then

$$\delta_j^{(1)} = o_j^{(1)}(1 - o_j^{(1)}) \sum_{q=1}^{m} w_{jq}^{(2)} \delta_q^{(2)}.$$

Therefore the partial derivative we are looking for is

$$\frac{\partial E}{\partial w_{ij}^{(1)}} = \delta_j^{(1)} o_i.$$

The backpropagated error can be computed in the same way for any number of hidden layers and the expression for the partial derivatives of E keeps the same analytic form.

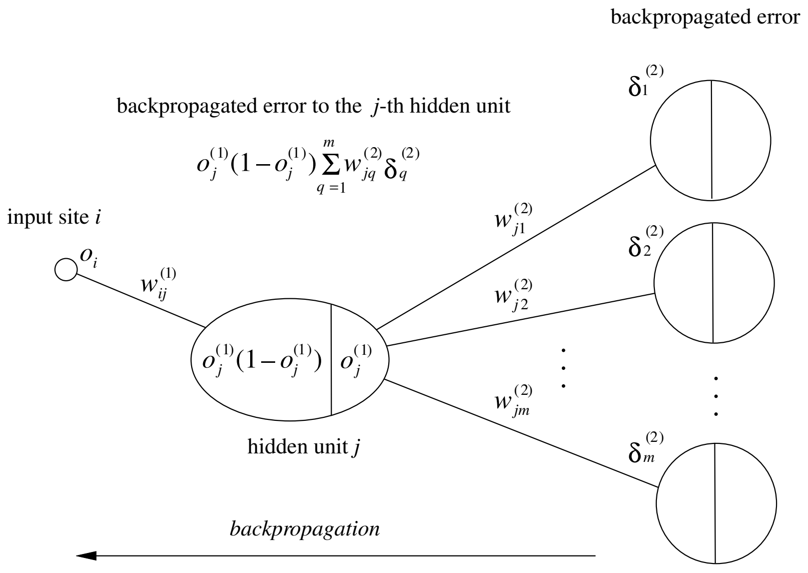


Fig. 7.21. All paths up to input site i

Fourth step: weight updates

After computing all partial derivatives the network weights are updated in the negative gradient direction. A learning constant γ defines the step length of the correction. The corrections for the weights are given by

$$\Delta w_{ij}^{(2)} = -\gamma o_i^{(1)} \delta_j^{(2)}, \quad \text{for } i = 1, \ldots, k+1; j = 1, \ldots, m,$$

and

$$\Delta w_{ij}^{(1)} = -\gamma o_i \delta_j^{(1)}, \quad \text{for } i = 1, \ldots, n+1; j = 1, \ldots, k,$$

where we use the convention that $o_{n+1} = o_{k+1}^{(1)} = 1$. It is very important to make the corrections to the weights only after the backpropagated error has been computed for all units in the network. Otherwise the corrections become intertwined with the backpropagation of the error and the computed corrections do not correspond any more to the negative gradient direction. Some authors fall in this trap [16]. Note also that some books define the backpropagated error as the *negative* traversing value in the network. In that case the update equations for the network weights do not have a negative sign (which is absorbed by the deltas), but this is a matter of pure convention.

More than one training pattern

In the case of $p > 1$ input-output patterns, an extended network is used to compute the error function for each of them separately. The weight corrections are computed for each pattern and so we get, for example, for weight $w_{ij}^{(1)}$ the corrections

$$\Delta_1 w_{ij}^{(1)}, \Delta_2 w_{ij}^{(1)}, \ldots, \Delta_p w_{ij}^{(1)}.$$

The necessary update in the gradient direction is then

$$\Delta w_{ij}^{(1)} = \Delta_1 w_{ij}^{(1)} + \Delta_2 w_{ij}^{(1)} + \cdots + \Delta_p w_{ij}^{(1)}.$$

We speak of *batch* or *off-line* updates when the weight corrections are made in this way. Often, however, the weight updates are made sequentially after each pattern presentation (this is called *on-line* training). In this case the corrections do not exactly follow the negative gradient direction, but if the training patterns are selected randomly the search direction oscillates around the exact gradient direction and, on average, the algorithm implements a form of descent in the error function. The rationale for using on-line training is that adding some noise to the gradient direction can help to avoid falling into shallow local minima of the error function. Also, when the training set consists of thousands of training patterns, it is very expensive to compute the exact gradient direction since each *epoch* (one round of presentation of all patterns to the network) consists of many feed-forward passes and on-line training becomes more efficient [391].

7.3.3 Backpropagation in matrix form

We have seen that the graph labeling approach for the proof of the backpropagation algorithm is completely general and is not limited to the case of regular layered architectures. However this special case can be put into a form suitable for vector processors or special machines for linear algebraic operations.

We have already shown that in a network with a hidden and an output layer (n, k and m units) the input $\mathbf{o}$ produces the output $\mathbf{o}^{(2)} = s(\hat{\mathbf{o}}^{(1)}\overline{\mathbf{W}}_2)$ where $\mathbf{o}^{(1)} = s(\hat{\mathbf{o}}\overline{\mathbf{W}}_1)$. In the backpropagation step we only need the first n rows of matrix $\overline{\mathbf{W}}_1$. We call this $n \times k$ matrix $\mathbf{W}_1$. Similarly, the $k \times m$ matrix $\mathbf{W}_2$ is composed of the first k rows of the matrix $\overline{\mathbf{W}}_2$. We make this reduction because we do not need to backpropagate any values to the constant inputs corresponding to each bias.

The derivatives stored in the feed-forward step at the k hidden units and the m output units can be written as the two diagonal matrices

$$\mathbf{D}_2 = \begin{pmatrix} o_1^{(2)}(1-o_1^{(2)}) & 0 & \cdots & 0 \\ 0 & o_2^{(2)}(1-o_2^{(2)}) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & o_m^{(2)}(1-o_m^{(2)}) \end{pmatrix},$$

and

$$\mathbf{D}_1 = \begin{pmatrix} o_1^{(1)}(1-o_1^{(1)}) & 0 & \cdots & 0 \\ 0 & o_2^{(1)}(1-o_2^{(1)}) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & o_k^{(1)}(1-o_k^{(1)}) \end{pmatrix}.$$

Define the vector $\mathbf{e}$ of the stored derivatives of the quadratic deviations as

$$\mathbf{e} = \begin{pmatrix} (o_1^{(2)} - t_1) \\ (o_2^{(2)} - t_2) \\ \vdots \\ (o_m^{(2)} - t_m) \end{pmatrix}$$

The m-dimensional vector $\boldsymbol{\delta}^{(2)}$ of the backpropagated error up to the output units is given by the expression

$$\boldsymbol{\delta}^{(2)} = \mathbf{D}_2\mathbf{e}.$$

The k-dimensional vector of the backpropagated error up to the hidden layer is

$$\boldsymbol{\delta}^{(1)} = \mathbf{D}_1\mathbf{W}_2\boldsymbol{\delta}^{(2)}.$$

The corrections for the matrices $\overline{\mathbf{W}}_1$ and $\overline{\mathbf{W}}_2$ are then given by

$$\Delta\overline{\mathbf{W}}_2^{\mathrm{T}} = -\gamma\boldsymbol{\delta}^{(2)}\hat{\mathbf{o}}^1 \tag{7.1}$$

and

$$\Delta\overline{\mathbf{W}}_1^{\mathrm{T}} = -\gamma\boldsymbol{\delta}^{(1)}\hat{\mathbf{o}}. \tag{7.2}$$

The only necessary operations are vector-matrix, matrix-vector, and vector-vector multiplications. In Chap. 16 we describe computer architectures optimized for this kind of operation. It is easy to generalize these equations for ℓ layers of computing units. Assume that the connection matrix between layer i and $i+1$ is denoted by $\overline{\mathbf{W}}_{i+1}$ (layer 0 is the layer of input sites). The backpropagated error to the output layer is then

$$\boldsymbol{\delta}^{(\ell)} = \mathbf{D}_\ell\mathbf{e}.$$

The backpropagated error to the i-th computing layer is defined recursively by

$$\boldsymbol{\delta}^{(i)} = \mathbf{D}_i\mathbf{W}_{i+1}\boldsymbol{\delta}^{(i+1)}, \qquad \text{for } i = 1, \ldots, \ell-1.$$

or alternatively

$$\boldsymbol{\delta}^{(i)} = \mathbf{D}_i\mathbf{W}_{i+1}\cdots\mathbf{W}_{\ell-1}\mathbf{D}_{\ell-1}\mathbf{W}_\ell\mathbf{D}_\ell\mathbf{e}.$$

The corrections to the weight matrices are computed in the same way as for two layers of computing units.

7.3.4 The locality of backpropagation

We can now prove using a B-diagram that addition is the only integration function which preserves the locality of learning when backpropagation is used.

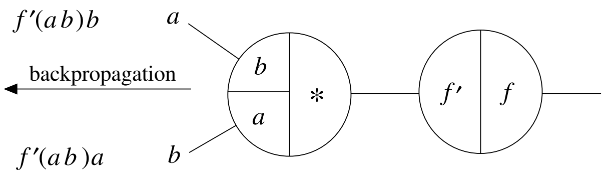


Fig. 7.22. Multiplication as integration function

In the networks we have seen so far the backpropagation algorithm exploits only local information. This means that only information which arrives from an input line in the feed-forward step is stored at the unit and sent back through the same line in the backpropagation step. An example can make this point clear. Assume that the integration function of a unit is multiplication and its activation function is f. Figure 7.22 shows a split view of the computation: two inputs a and b come from two input lines, the integration function responds with the product ab and this result is passed as argument to f. With backpropagation we can compute the partial derivative of $f(ab)$ with respect to a and with respect to b. But in this case the value b must be transported back through the upper edge and the value a through the lower one. Since b arrived through the other edge, the locality of the learning algorithm has been lost. The question is which kinds of integration functions preserve the locality of learning. The answer is given by the following proposition.

Proposition 12. *In a unit with n inputs $x_1, \ldots, x_n$ only integration functions of the form*

$$I(x_1, \ldots, x_n) = F_1(x_1) + F_2(x_2) + \cdots + F_n(x_n) + C,$$

where C is a constant, guarantee the locality of the backpropagation algorithm in the sense that at an edge $i \neq j$ no information about x_j has to be explicitly stored.

Proof. Let the integration function of the unit be the function I of n arguments. If, in the backpropagation step, only a function f_i of the variable x_i can be stored at the computing unit in order to be transmitted through the i-th input line in the backpropagation step, then we know that

$$\frac{\partial I}{\partial x_i} = f_i(x_i), \quad \text{for } i = 1, \ldots, n.$$

Therefore by integrating these equations we obtain:

$$\begin{aligned} I(x_1, x_2, \ldots, x_n) &= F_1(x_1) + G_1(x_2, \ldots, x_n), \\ I(x_1, x_2, \ldots, x_n) &= F_2(x_2) + G_2(x_2, \ldots, x_n), \\ &\vdots \\ I(x_1, x_2, \ldots, x_n) &= F_n(x_n) + G_n(x_2, \ldots, x_n), \end{aligned}$$

where F_i denotes the integral of f_i and $G_1, \ldots, G_n$ are real functions of $n-1$ arguments. Since the function I has a unique form, the only possibility is

$$I(x_1, x_2, \ldots, x_n) = F_1(x_1) + F_2(x_2) + \cdots + F_n(x_n) + C$$

where C is a constant. This means that information arriving from each line can be preprocessed by the F_i functions and then has to be added. Therefore only integration functions with this form preserve locality. □

7.3.5 Error during training

We discussed the form of the error function for the XOR problem in the last chapter. It is interesting to see how backpropagation performs when confronted with this problem. Figure 7.23 shows the evolution of the total error during training of a network of three computing units. After 600 iterations the algorithm found a solution to the learning problem. In the figure the error falls fast at the beginning and end of training. Between these two zones lies a region in which the error function seems to be almost flat and where progress is slow. This corresponds to a region which would be totally flat if step functions were used as activation functions of the units. Now, using the sigmoid, this region presents a small slope in the direction of the global minimum.

In the next chapter we discuss how to make backpropagation converge faster, taking into account the behavior of the algorithm at the flat spots of the error function.

7.4 Recurrent networks

The backpropagation algorithm can also be extended to the case of recurrent networks. To deal with this kind of systems we introduce a discrete time variable t. At time t all units in the network recompute their outputs, which are then transmitted at time $t+1$. Continuing in this step-by-step fashion, the system produces a sequence of output values when a constant or time varying input is fed into the network. As we already saw in Chap. 2, a recurrent network behaves like a finite automaton. The question now is how to train such an automaton to produce a desired sequence of output values.

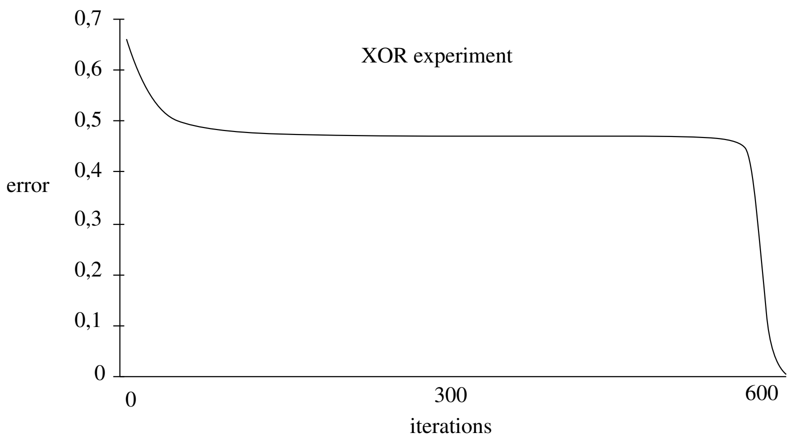


Fig. 7.23. Error function for 600 iterations of backpropagation

7.4.1 Backpropagation through time

The simplest way to deal with a recurrent network is to consider a finite number of iterations only. Assume for generality that a network of n computing units is fully connected and that w_{ij} is the weight associated with the edge from node i to node j. By unfolding the network at the time steps $1, 2, \ldots, T$, we can think of this recurrent network as a feed-forward network with T stages of computation. At each time step t an external input $\mathbf{x}(t)$ is fed into the network and the outputs $(o_1^{(t)}, \ldots, o_n^{(t)})$ of all computing units are recorded. We call the n-dimensional vector of the units' outputs at time t the network state $\mathbf{o}^{(t)}$. We assume that the initial values of all unit's outputs are zero at $t = 0$, but the external input $\mathbf{x}(0)$ can be different from zero. Figure 7.24 shows a diagram of the unfolded network. This unfolding strategy which converts a recurrent network into a feed-forward network in order to apply the backpropagation algorithm is called *backpropagation through time* or just BPTT [383].

Let $\mathbf{W}$ stand for the $n \times n$ matrix of network weights w_{ij}. Let $\mathbf{W}_0$ stand for the $m \times n$ matrix of interconnections between m input sites and n units. The feed-forward step is computed in the usual manner, starting with an initial m-dimensional external input $\mathbf{x}^{(0)}$. At each time step t the network state $\mathbf{o}^{(t)}$ (an n-dimensional row vector) and the vector of derivatives of the activation function at each node $\mathbf{o}'^{(t)}$ are stored. The error of the network can be measured after each time step if a sequence of values is to be produced, or just after the final step T if only the final output is of importance. We will handle the first, more general case. Denote the difference between the n-dimensional target $\mathbf{y}^{(t)}$ at time t and the output of the network by $\mathbf{e}^{(t)} = \left(\mathbf{o}^{(t)} - \mathbf{y}^{(t)}\right)^{\mathrm{T}}$. This is an n-dimensional column vector, but in most cases we are only interested in the outputs of some units in the network. In that case

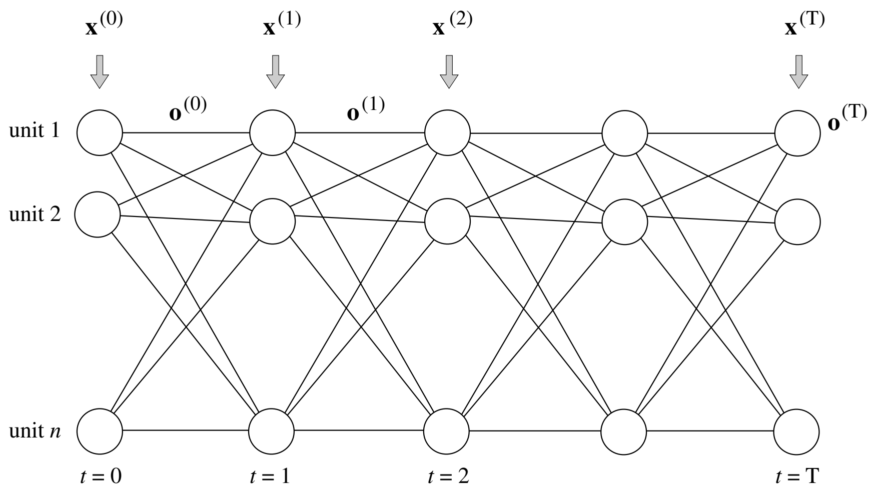


Fig. 7.24. Backpropagation through time

define $e_i(t) = 0$ for each unit i, whose precise state is unimportant and which can remain hidden from view.

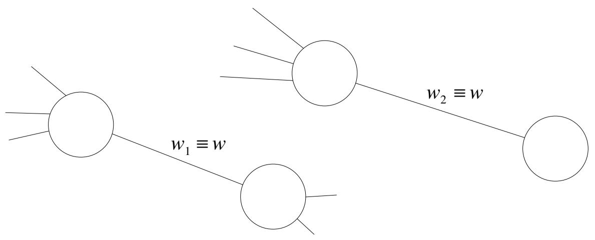


Fig. 7.25. A duplicated weight in a network

Things become complicated when we consider that each weight in the network is present at each stage of the unfolded network. Until now we had only handled the case of unique weights. However, any network with repeated weights can easily be transformed into a network with unique weights. Assume that after the feed-forward step the state of the network is the one shown in Figure 7.25. Weight w is duplicated, but received different inputs o_1 and o_2 in the feed-forward step at the two different locations in the network. The transformed network in Figure 7.26 is indistinguishable from the original network from the viewpoint of the results it produces. Note that the two edges associated with weight w now have weight 1 and a multiplication is performed by the two additional units in the middle of the edges. In this transformed network w appears only once and we can perform backpropagation as usual. There are two groups of paths, the ones coming from the first multiplier to w

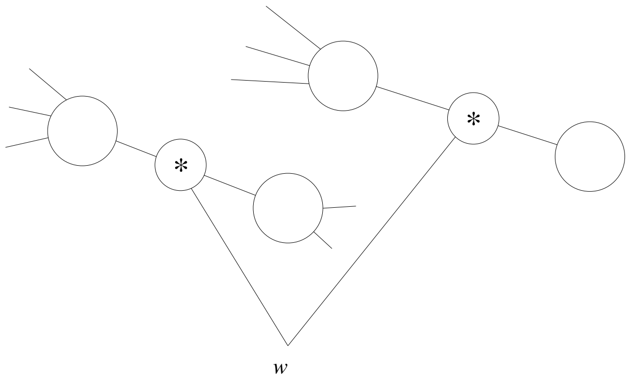


Fig. 7.26. Transformed network

and the ones coming from the second. This means that we can just perform backpropagation as usual in the original network. At the first edge we obtain $\partial E/\partial w_1$, at the second $\partial E/\partial w_2$, and since w_1 is the same variable as w_2, the desired partial derivative is

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial w_1} + \frac{\partial E}{\partial w_2}.$$

We can thus conclude in general that in the case of the same weight being associated with several edges, backpropagation is performed as usual for each of those edges and the results are simply added.

The backpropagation phase of BPTT starts from the right. The backpropagated error at time T is given by

$$\boldsymbol{\delta}^{(T)} = \mathbf{D}^{(T)}\mathbf{e}^{(T)},$$

where $\mathbf{D}^{(T)}$ is the $n \times n$ diagonal matrix whose component at the i-th diagonal element is $o_i'^{(T)}$, i.e., the stored derivative of the i-th unit output at time T. The backpropagated error at time $T-1$ is given by

$$\boldsymbol{\delta}^{(T-1)} = \mathbf{D}^{(T-1)}\mathbf{e}^{(T-1)} + \mathbf{D}^{(T-1)}\mathbf{W}\mathbf{D}^{(T)}\mathbf{e}^{(T)}\,,$$

where we have considered all paths from the computed errors at time T and $T-1$ to each weight. In general, the backpropagated error at stage i, for $i = 0, \ldots, T-1$ is

$$\boldsymbol{\delta}^{(i)} = \mathbf{D}^{(i)}(\mathbf{e}^{(i)} + \mathbf{W}\boldsymbol{\delta}^{(i+1)}).$$

The analytic expression for the final weight corrections are

$$\Delta\overline{\mathbf{W}}^T = -\gamma\left(\boldsymbol{\delta}^{(1)}\hat{\mathbf{o}}^{(0)} + \cdots + \boldsymbol{\delta}^{(T)}\hat{\mathbf{o}}^{(T-1)}\right) \tag{7.3}$$

$$\Delta\overline{\mathbf{W}}_0^T = -\gamma\left(\boldsymbol{\delta}^{(0)}\hat{\mathbf{x}}^{(0)} + \cdots + \boldsymbol{\delta}^{(T)}\hat{\mathbf{x}}^{(T)}\right), \tag{7.4}$$

where $\hat{\mathbf{o}}^{(1)}, \ldots, \hat{\mathbf{o}}^{(T)}$ denote the extended output vectors at steps $1, \ldots, T$ and $\overline{\mathbf{W}}$ and $\overline{\mathbf{W}}_0$ the extended matrices $\mathbf{W}$ and $\mathbf{W}_0$.

Backpropagation through time can be extended to the case of infinite time T. In this case we are interested in the limit value of the network's state, on the assumption that the network's state stabilizes to a fixpoint $\tilde{\mathbf{o}}$. Under some conditions over the activation functions and network topology such a fixpoint exists and its derivative can be calculated by backpropagation. The feed-forward step is repeated a certain number of times, until the network relaxes to a numerically stable state (with certain precision). The stored node's outputs and derivatives are the ones computed in the last iteration. The network is then run backwards in backpropagation manner until it reaches a numerically stable state. The gradient of the network function with respect to each weight can then be calculated in the usual manner. Note that in this case, we do not need to store all intermediate values of outputs and derivatives at the units, only the final ones. This algorithm, called recurrent backpropagation, was proposed independently by Pineda [342] and Almeida [20].

7.4.2 Hidden Markov Models

Hidden Markov Models (HMM) form an important special type of recurrent network. A first-order Markov model is any system capable of assuming one of n different states at time t. The system does not change its state at each time step deterministically but according to a stochastic dynamics. The probability of transition from the i-th to the j-th state at each step is given by $0 \leq a_{ij} \leq 1$ and does not depend on the previous history of transitions. These probabilities can be arranged in an $n \times n$ matrix A. We also assume that at each step the model emits one of m possible output values. We call the probability of emitting the k-th output value while in the i-th state b_{ik}. Starting from a definite state at time $t = 0$, the system is allowed to run for T time units and the generated outputs are recorded. Each new run of the system generally produces a different sequence of output values. The system is called a HMM because only the emitted values, not the state transitions, can be observed.

An example may make this point clear. In speech recognition researchers postulate that the vocal tract shapes can be quantized in a discrete set of states roughly associated with the phonemes which compose speech. When speech is recorded the exact transitions in the vocal tract cannot be observed and only the produced sound can be measured at some predefined time intervals. These are the emissions, and the states of the system are the quantized configurations of the vocal tract. From the measurements we want to infer the sequence of states of the vocal tract, i.e., the sequence of utterances which gave rise to the recorded sounds. In order to make this problem manageable, the set of states and the set of possible sound parameters are quantized (see Chap. 9 for a deeper discussion of automatic speech recognition).

The general problem when confronted with the recorded sequence of output values of a HMM is to compute the most probable sequence of state

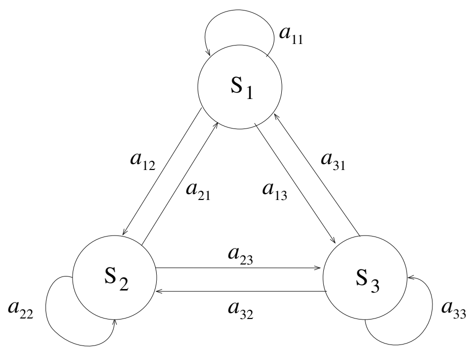


Fig. 7.27. Transition probabilities of a Markov model with three states

transitions which could have produced them. This is done with a recursive algorithm.

The state diagram of a HMM can be represented by a network made of n units (one for each state) and with connections from each unit to each other. The weights of the connections are the transition probabilities (Figure 7.27).

As in the case of backpropagation through time, we can unfold the network in order to observe it at each time step. At $t = 0$ only one of the n units, say the i-th, produces the output 1, all others zero. State i is the the actual state of the system. The probability that at time $t = 1$ the system reaches state j is given by a_{ij} (to avoid cluttering only some of these values are shown in the diagram). The probability of reaching state k at $t = 2$ is

$$\sum_{j=1}^{n} a_{ij}a_{jk}$$

which is just the net input at the k-th node in the stage $t = 2$ of the network shown in Figure 7.28. Consider now what happens when we can only observe the output of the system but not the state transitions (refer to Figure 7.29). If the system starts at $t = 0$ in a state given by a discrete probability distribution $\rho_1, \rho_2, \ldots, \rho_n$, then the probability of observing the k-th output at $t = 0$ is given by

$$\sum_{i=1}^{n} \rho_i b_{ik}.$$

The probability of observing the k-th output at $t = 0$ *and* the m-th output at $t = 1$ is

$$\sum_{j=1}^{n} \sum_{i=1}^{n} \rho_i b_{ik} a_{ij} b_{jm}.$$

The rest of the stages of the network compute the corresponding probabilities in a similar manner.

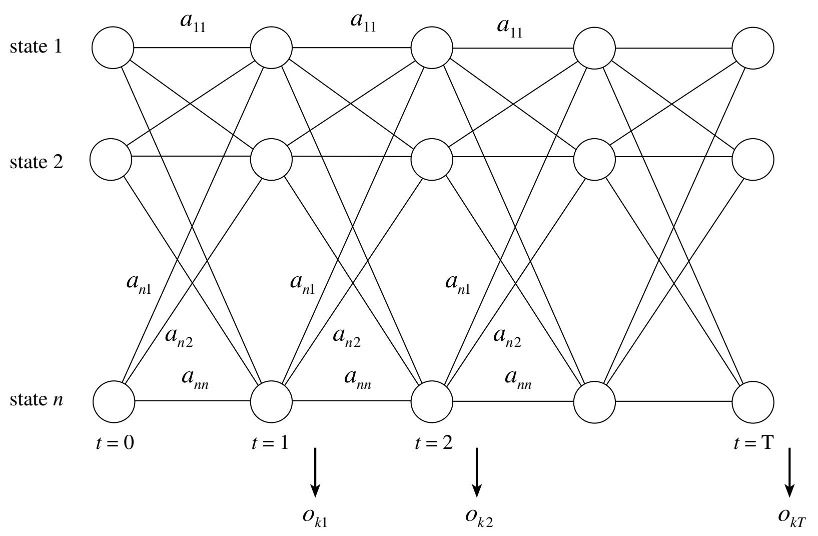


Fig. 7.28. Unfolded Hidden Markov Model

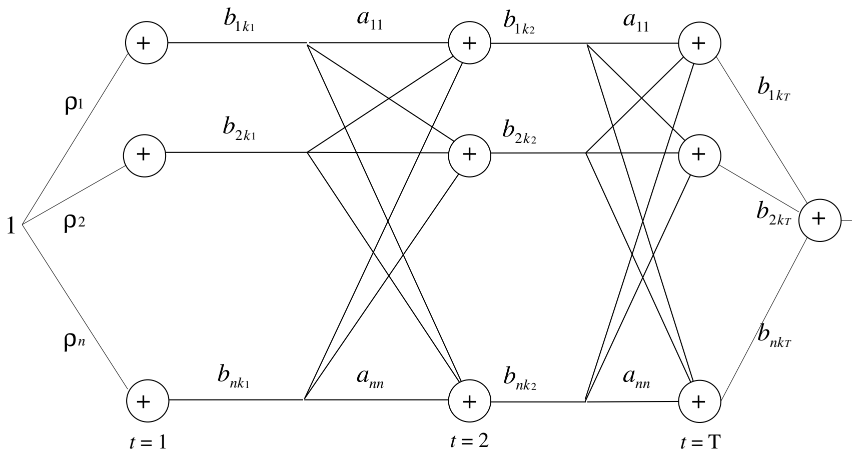


Fig. 7.29. Computation of the likelihood of a sequence of observations

How can we find the unknown transition and emission probabilities for such an HMM? If we are given a sequence of T observed outputs with indices $k_1, k_2, \ldots, k_T$ we would like to maximize the likelihood of this sequence, i.e., the product of the probabilities that each of them occurs. This can be done by transforming the unfolded network as shown in Figure 7.29 for $T = 3$. Notice that at each stage h we introduced an additional edge from the node i with the weight b_{i,k_h}. In this way the final node which collects the sum of the whole computation effectively computes the likelihood of the observed sequence.

Since this unfolded network contains only differentiable functions at its nodes (in fact only addition and the identity function) it can be trained using the backpropagation algorithm. However, care must be taken to avoid updating the probabilities in such a way that they become negative or greater than 1. Also the transition probabilities starting from the same node must always add up to 1. These conditions can be enforced by an additional transformation of the network (introducing for example a "softmax" function [39]) or using the method of Lagrange multipliers. We give only a hint of how this last technique can be implemented so that the reader may complete the network by her- or himself.

Assume that a function F of n parameters $x_1, x_2, \ldots, x_n$ is to be minimized, subject to the constraint $C(x_1, x_2, \ldots, x_n) = 0$. We introduce a Lagrange multiplier λ and define the new function

$$L(x_1, \ldots, x_n, \lambda) = F(x_1, \ldots, x_n) + \lambda C(x_1, \ldots, x_n).$$

To minimize L we compute its gradient and set it to zero. To do this numerically, we follow the negative gradient direction to find the minimum. Note that since

$$\frac{\partial L}{\partial \lambda} = C(x_1, \ldots, x_n)$$

the iteration process does not finish as long as $C(x_1, \ldots, x_n) \neq 0$, because in that case the partial derivative of L with respect to λ is non-zero. If the iteration process converges, we can be sure that the constraint C is satisfied. Care must be taken when the minimum of F is reached at a saddle point of L. In this case some modifications of the basic gradient descent algorithm are needed [343]. Figure 7.30 shows a diagram of the network (a *Lagrange neural network* [468]) adapted to include a constraint. Since all functions in the network are differentiable, the partial derivatives needed can be computed with the backpropagation algorithm.

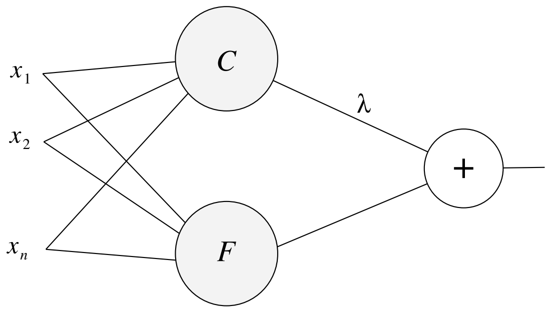


Fig. 7.30. Lagrange neural network

7.4.3 Variational problems

Our next example, deals not with a recurrent network, but with a class of networks built of many repeated stages. Variational problems can also be expressed and solved numerically using backpropagation networks. A variational problem is one in which we are looking for a function which can optimize a certain cost function. Usually cost is expressed analytically in terms of the unknown function and finding a solution is in many cases an extremely difficult problem. An example can illustrate the general technique that can be used.

Assume that the problem is to minimize P with two boundary conditions:

$$P(u) = \int_o^1 F(u,u')dx \qquad \text{with } u(0) = a \quad \text{and} \quad u(1) = b.$$

Here u is an unknown function of x and $F(u,u')$ a cost function. P represents the total cost associated with u over the interval $[0,1]$. Since we want to solve this problem numerically, we discretize the function u by partitioning the interval $[0,1]$ into $n-1$ subintervals. The discrete successive values of the function are denoted by $u_1, u_2, \ldots, u_n$, where $u_1 = a$ and $u_n = b$ are the boundary conditions. The length of the subintervals is $\Delta x = 1/(n-1)$. The discrete function P_d that we want to minimize is thus:

$$P_d(u) = \sum_{i=1}^{n} F(u_i, u_i')\Delta x.$$

Since minimizing $P_d(u)$ is equivalent to minimizing $P_D(u) = P_d(u)/\Delta x$ (Δx is constant), we proceed to minimize $P_D(u)$. We can approximate the derivative u_i' by $(u_i - u_{i-1})/\Delta x$. Figure 7.31 shows the network that computes the discrete approximation $P_D(u)$.

We can now compute the gradient of P_D with respect to each u_i by performing backpropagation on this network. Note that there are three possible paths from P_D to u_i, so the partial derivative of P_D with respect to u_i is

$$\frac{\partial P_D}{\partial u_i} = \underbrace{\frac{\partial F}{\partial u}(u_i,u_i')}_{path1} + \underbrace{\frac{1}{\Delta x}\frac{\partial F}{\partial u'}(u_i,u_i')}_{path2} - \underbrace{\frac{1}{\Delta x}\frac{\partial F}{\partial u'}(u_{i+1},u_{i+1}')}_{path3}$$

which can be rearranged to

$$\frac{\partial P_d}{\partial u_i} = \frac{\partial F}{\partial u}(u_i,u_i') - \frac{1}{\Delta x}\left(\frac{\partial F}{\partial u'}(u_{i+1},u_{i+1}') - \frac{\partial F}{\partial u'}(u_i,u_i')\right).$$

At the minimum all these terms should vanish and we get the expression

$$\frac{\partial F}{\partial u}(u_i,u_i') - \frac{1}{\Delta x}\left(\frac{\partial F}{\partial u'}(u_{i+1},u_{i+1}') - \frac{\partial F}{\partial u'}(u_i,u_i')\right) = 0$$

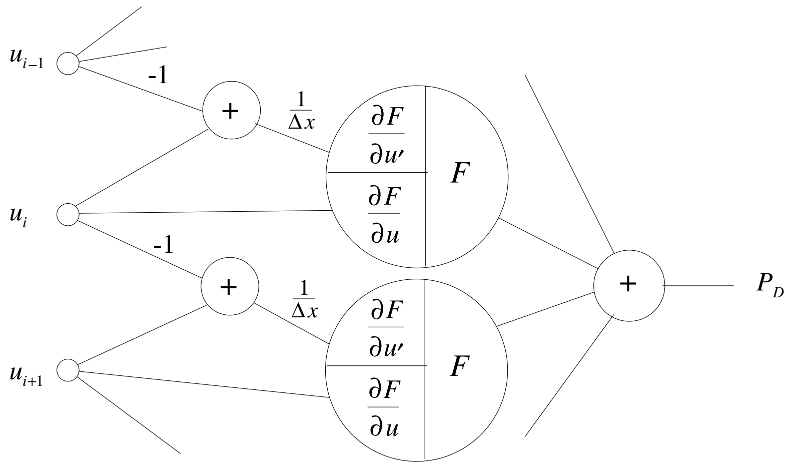

u_{i-1}
-1
+
$\frac{1}{\Delta x}$
$\frac{\partial F}{\partial u'}$
$\frac{\partial F}{\partial u}$
F
u_i
-1
+
$\frac{1}{\Delta x}$
$\frac{\partial F}{\partial u'}$
$\frac{\partial F}{\partial u}$
F
u_{i+1}
+
P_D

Fig. 7.31. A network for variational calculus

which is the discrete version of the celebrated Euler equation

$$\frac{\partial F}{\partial u} - \frac{d}{dx}\left(\frac{\partial F}{\partial u'}\right) = 0.$$

In fact this can be considered a simple derivation of Euler's result in a discrete setting.

By selecting another function F many variational problems can be solved numerically. The curve of minimal length between two points can be found by using the function

$$F(u,u') = \sqrt{1+u'^2} = \frac{\sqrt{dx^2+du^2}}{dx}$$

which when integrated with respect to x corresponds to the path length between the boundary conditions. In 1962 Dreyfus solved the constrained brachystochrone problem, one of the most famous problems of variational calculus, using a numerical approach similar to the one discussed here [115].

7.5 Historical and bibliographical remarks

The field of neural networks started with the investigations of researchers of the caliber of McCulloch, Wiener, and von Neumann. The perceptron era was its *Sturm und Drang* period, the epoch in which many new ideas were tested and novel problems were being solved using perceptrons. However, at the end of the 1960s it became evident that more complex multilayered architectures demanded a new learning paradigm. In the absence of such an algorithm, a new era of cautious incremental improvements and silent experimentation began.

The algorithm that the neural network community needed had already been developed by researchers working in the field of optimal control. These researchers were dealing with variational problems with boundary conditions in which a function capable of optimizing a cost function subject to some constraints must be determined. As in the field of neural networks, a function f must be found and a set of input-output values is predefined for some points. Around 1960 Kelley and Bryson developed numerical methods to solve this variational problem which relied on a recursive calculation of the gradient of a cost function according to its unknown parameters [241, 76]. In 1962 Dreyfus, also known for his criticism of symbolic AI, showed how to express the variational problems as a kind of multistage system and gave a simple derivation of what we now call the backpropagation algorithm [115, 116]. He was the first to use an approach based on the chain rule, in fact one very similar to that used later by the PDP group. Bryson and Ho later summarized this technique in their classical book on optimal control [76]. However, Bryson gives credit for the idea of using numerical methods to solve variational problems to Courant, who in 1943 proposed using gradient descent along the Euler expression (the partial derivative of the cost function) to find numerical approximations to variational problems [93].

The algorithm was redeveloped by some other researchers working in the field of statistics or pattern recognition. We can look as far back as Gauss to find mathematicians already doing function-fitting with numerical methods. Gauss developed the method of least squares and considered the fitting of nonlinear functions of unknown parameters. In the Gauss–Newton method the function F of parameters $w_1, \ldots, w_n$ is approximated by its Taylor expansion at an initial point using only the first-order partial derivatives. Then a least-squares problem is solved and a new set of parameters is found. This is done iteratively until the function F approximates the empirical data with the desired accuracy [180]. Another possibility, however, is the use of the partial derivatives of F with respect to the parameters to do a search in the gradient direction. This approach was already being used by statisticians in the 1960s [292]. In 1974 Werbos considered the case of general function composition and proposed the backpropagation algorithm [442, 443] as a kind of nonlinear regression. The points given as the training set are considered not as boundary conditions, which cannot be violated, but as experimental points which have to be approximated by a suitable function. The special case of recursive backpropagation for Hidden Markov Models was solved by Baum, also considering the restrictions on the range of probability values [47], which he solved by doing a projective transformation after each update of the set of probabilities. His "forward-backward" algorithm for HMMs can be considered one of the precursors of the backpropagation algorithm.

Finally, the AI community also came to rediscovering backpropagation on its own. Rumelhart and his coauthors [383] used it to optimize multilayered neural networks in the 1980s. Le Cun is also mentioned frequently as one of the authors who reinvented backpropagation [269]. The main difference however to the approach of both the control or statistics community was in conceiving the networks of functions as interconnected computing units. We said before that backpropagation reduces to the recursive computation of the chain rule. But there is also a difference: the network of computing units serves as the underlying data structure to store values of previous computations, avoiding redundant work by the simple expedient of running the network backwards and labeling the nodes with the backpropagated error. In this sense the backpropagation algorithm, as rediscovered by the AI community, added something new, namely the concept of functions as dynamical objects being evaluated by a network of computing elements and backpropagation as an inversion of the network dynamics.

Exercises

1. Implement the backpropagation algorithm and train a network that computes the parity of 5 input bits.
2. The symmetric sigmoid is defined as $t(x) = 2s(x) - 1$, where $s(\cdot)$ is the usual sigmoid function. Find the new expressions for the weight corrections in a layered network in which the nodes use t as a primitive function.
3. Find the analytic expressions for the partial derivative of the error function according to each one of the input values to the network. What could be done with this kind of information?
4. Find a discrete approximation to the curve of minimal length between two points in $\mathbb{R}^3$ using a backpropagation network.

8 Fast Learning Algorithms

8.1 Introduction – classical backpropagation

Artificial neural networks attracted renewed interest over the last decade, mainly because new learning methods capable of dealing with large scale learning problems were developed. After the pioneering work of Rosenblatt and others, no efficient learning algorithm for multilayer or arbitrary feed-forward neural networks was known. This led some to the premature conclusion that the whole field had reached a dead-end. The rediscovery of the backpropagation algorithm in the 1980s, together with the development of alternative network topologies, led to the intense outburst of activity which put neural computing back into the mainstream of computer science.

Much has changed since the original proposals of the PDP group. There is now no lack of alternative fast learning algorithms for neural networks. Each new conference and each new journal issue features some kind of novel learning method offering better and faster convergence than the tried and trusted standard backpropagation method. The reason for this combinatorial explosion of new algorithms is twofold: on the one hand, backpropagation itself is a rather *slow* learning algorithm, which through malicious selection of parameters can be made even slower. By using any of the well-known techniques of nonlinear optimization, it is possible to accelerate the training method with practically no effort. Since authors usually compare their new algorithms with standard backpropagation, they always report a substantial improvement [351]. On the other hand, since the learning problem for artificial neural networks is *NP*-complete (see Chap. 10) in the worst case, the computational effort involved in computing the network parameters grows exponentially with the number of unknowns. This leaves room for alternative proposals which could deal with some learning tasks in a more efficient manner. However, it is always possible to fool the best learning method with a suitable learning task and it is always possible to make it perform incomparably better than all its competitors. It is a rather surprising fact that standard on-line backpropagation performs better than many fast learning algorithms as soon as the learning task achieves a realistic level of complexity and when the size of the training set goes beyond a critical threshold [391].

In this chapter we try to introduce some order into the burgeoning field of fast learning algorithms for neural networks. We show the essential characteristics of the proposed methods, the fundamental alternatives open to anyone wishing to improve traditional backpropagation and the similarities and differences between the different techniques. Our analysis will be restricted to those algorithms which deal with a fixed network topology. One of the lessons learned over the past years is that significant improvements in the approximation capabilities of neural networks will only be obtained through the use of *modularized networks.* In the future, more complex learning algorithms will deal not only with the problem of determining the network parameters, but also with the problem of adapting the network topology. Algorithms of this type already in existence fall beyond the scope of this chapter.

8.1.1 Backpropagation with momentum

Before reviewing some of the variations and improvements which have been proposed to accelerate the learning process in neural networks, we briefly discuss the problems involved in trying to minimize the error function using gradient descent. Therefore, we first describe a simple modification of backpropagation called *backpropagation with momentum*, and then look at the form of the iteration path in weight space.

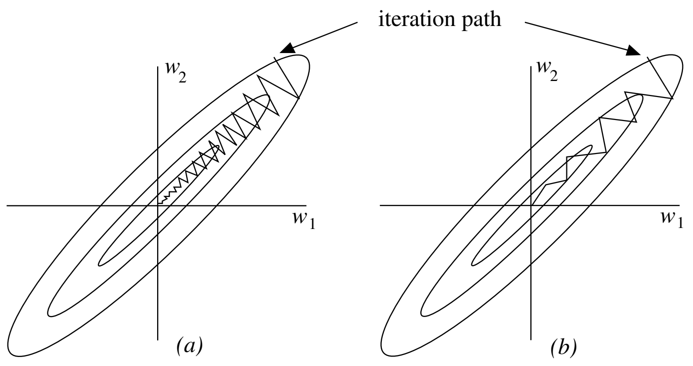


Fig. 8.1. Backpropagation without (a) or with (b) momentum term

When the minimum of the error function for a given learning task lies in a narrow "valley", following the gradient direction can lead to wide oscillations of the search process. Figure 8.1 shows an example for a network with just two weights w_1 and w_2. The best strategy in this case is to orient the search towards the center of the valley, but the form of the error function is such that the gradient does not point in this direction. A simple solution is to introduce a *momentum* term. The gradient of the error function is computed for each new combination of weights, but instead of just following the negative gradient direction a *weighted average* of the current gradient and the previous correction direction is computed at each step. Theoretically, this approach should provide the search process with a kind of *inertia* and could help to avoid excessive oscillations in narrow valleys of the error function.

As explained in the previous chapter, in standard backpropagation the input-output patterns are fed into the network and the error function E is determined at the output. When using backpropagation with momentum in a network with n different weights $w_1, w_2, \ldots, w_n$, the i-th correction for weight w_k is given by

$$\Delta w_k(i) = -\gamma \frac{\partial E}{\partial w_k} + \alpha \Delta w_k(i-1),$$

where γ and α are the learning and momentum rate respectively. Normally, we are interested in accelerating convergence to a minimum of the error function, and this can be done by increasing the learning rate up to an optimal value. Several fast learning algorithms for neural networks work by trying to find the best value of γ which still guarantees convergence. Introduction of the momentum rate allows the attenuation of oscillations in the iteration process.

The adjustment of both learning parameters to obtain the best possible convergence is normally done by trial and error or even by some kind of random search [389]. Since the optimal parameters are highly dependent on the learning task, no general strategy has been developed to deal with this problem. Therefore, in the following we show the trade-offs involved in choosing a specific learning and momentum rate, and the kind of oscillating behavior which can be observed with the backpropagation feedback rule and large momentum rates. We show that they are necessary when the optimal size of the learning step is unknown and the form of the error function is highly degenerate.

The linear associator

Let us first consider the case of a *linear associator*, that is, a single computing element with associated weights $w_1, w_2, \ldots, w_n$ and which for the input $x_1, x_2, \ldots, x_n$ produces $w_1x_1 + \cdots + w_nx_n$ as output, as shown in Figure 8.2.

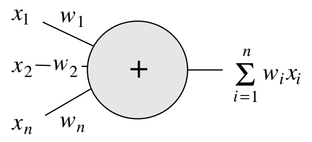


Fig. 8.2. Linear associator

The input-output patterns in the training set are the p ordered pairs $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_p, y_p)$, whereby the input patterns are row vectors of dimension n and the output patterns are scalars. The weights of the linear associator can be ordered in an n-dimensional column vector $\mathbf{w}$ and the learning task consists of finding the $\mathbf{w}$ that minimizes the quadratic error

$$E = \sum_{i=1}^{n} \|\mathbf{x}_i \cdot \mathbf{w} - y_i\|^2.$$

By defining a $p \times m$ matrix $\mathbf{X}$ whose rows are the vectors $\mathbf{x}_1, \ldots, \mathbf{x}_p$ and a column vector $\mathbf{y}$ whose elements are the scalars $y_1, \ldots, y_p$, the learning task reduces to the minimization of

$$\begin{aligned} E &= \|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2 \\ &= (\mathbf{X}\mathbf{w} - \mathbf{y})^{\mathrm{T}}(\mathbf{X}\mathbf{w} - \mathbf{y}) \\ &= \mathbf{w}^{\mathrm{T}}(\mathbf{X}^{\mathrm{T}}\mathbf{X})\mathbf{w} - 2\mathbf{y}^{\mathrm{T}}\mathbf{X}\mathbf{w} + \mathbf{y}^{\mathrm{T}}\mathbf{y}. \end{aligned}$$

Since this is a quadratic function, the minimum can be found using gradient descent.

The quadratic function E can be thought of as a paraboloid in m-dimensional space. The lengths of its principal axes are determined by the magnitude of the eigenvalues of the correlation matrix $\mathbf{X}^{\mathrm{T}}\mathbf{X}$. Gradient descent is most effective when the principal axes of the quadratic form are all of the same length. In this case the gradient vector points directly towards the minimum of the error function. When the axes of the paraboloid are of very different sizes, the gradient direction can lead to oscillations in the iteration process as shown in Figure 8.1.

Let us consider the simple case of the quadratic function $ax^2 + by^2$. Gradient descent with momentum yields the iteration rule

$$\Delta x(i) = -2\gamma a x + \alpha \Delta x(i-1)$$

in the x direction and

$$\Delta y(i) = -2\gamma b x + \alpha \Delta y(i-1)$$

in the y direction. An optimal parameter combination in the x direction is $\gamma = 1/2a$ and $\alpha = 0$. In the y direction the optimal combination is $\gamma = 1/2b$ and $\alpha = 0$. Since iteration proceeds with a single γ value, we have to find a compromise between these two options. Intuitively, if the momentum term is zero, an intermediate γ should do best. Figure 8.3 shows the number of iterations needed to find the minimum of the error function to a given precision as a function of γ, when $a = 0.9$ and $b = 0.5$. The optimal value for γ is the one found at the intersection of the two curves and is $\gamma = 0.7$. The global optimal γ is larger than the optimal γ in the x direction and smaller than the optimal γ in the y direction. This means that there will be some oscillations

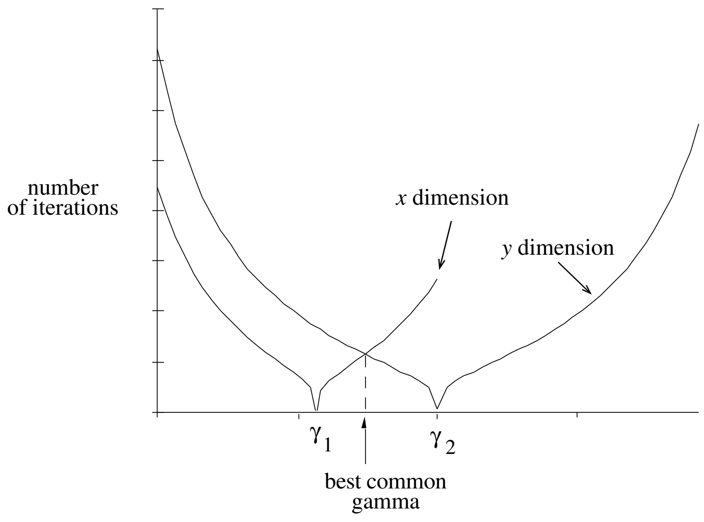


Fig. 8.3. Optimal γ in the two-dimensional case

in the y direction and slow convergence in the x direction, but this is the best possible compromise. It is obvious that in the n-dimensional case we could have oscillations in some of the principal directions and slow convergence in others. A simple strategy used by some fast learning algorithms to avoid these problems consists of using a different learning rate for each weight, that is, a different γ for each direction in weight space [217].

Minimizing oscillations

Since the lengths of the principal axes of the error function are given by the eigenvalues of the correlation matrix $\mathbf{X}^{\mathrm{T}}\mathbf{X}$, and since one of these eigenvalues could be much larger than the others, the range of possible values for γ reduces accordingly. Nevertheless, a very small γ and the oscillations it produces can be neutralized by increasing the momentum rate. We proceed to a detailed discussion of the one-dimensional case, which provides us with the necessary insight for the more complex multidimensional case.

In the one-dimensional case, that is, when minimizing functions of type kx^2, the optimal learning rate is given by $\gamma = 1/2k$. The rate $\gamma = 1/k$ produces an oscillation between the initial point x_0 and $-x_0$. Any γ greater than $2/k$ leads to an "explosion" of the iteration process. Figure 8.4 shows the main regions for parameter combinations of γ and the momentum rate α. These regions were determined by iterating in the one-dimensional case and *integrating* the length of the iteration path. Parameter combinations in the divergence region lead to divergence of the iteration process. Parameter combinations in the boundary between regions lead to stable oscillations.

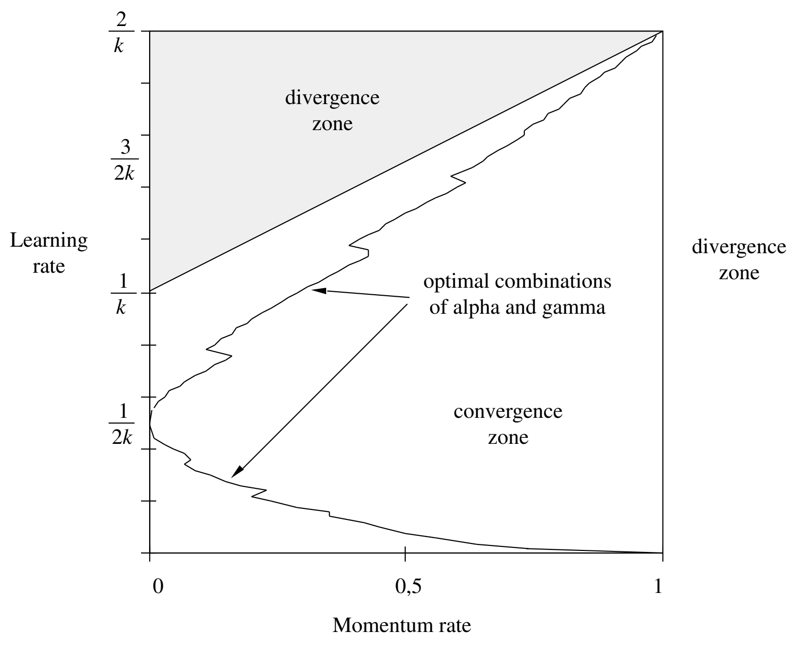


Fig. 8.4. Convergence zone for combinations of γ and α

Figure 8.4 shows some interesting facts. Any value of γ greater than four times the constant $1/2k$ cannot be balanced with any value of α. Values of α greater than 1 are prohibited since they lead to a geometric explosion of the iteration process. Any value of γ between the explosion threshold $1/k$ and $2/k$ can lead to convergence if a large enough α is selected. For any given γ between $1/k$ and $2/k$ there are two points at which the iteration process falls in a stable oscillation, namely at the boundaries between regions. For values of γ under the optimal value $1/2k$, the convergence speed is optimal for a unique α. The optimal combinations of α and γ are the ones represented by the jagged line in the diagram.

The more interesting message we get from Figure 8.4 is the following: in the case where in some direction in weight space the principal axis of the error function is very small compared to another axis, we should try to achieve a compromise by adjusting the momentum rate in such a way that the directions in which the algorithm oscillates become less oscillating and the directions with slow convergence improve their convergence speed. Obviously when dealing with n axes in weight space, this compromise could be dominated by a single direction.

Critical parameter combinations

Backpropagation is used in those cases in which we do not have an analytic expression of the error function to be optimized. A learning rate γ has to be chosen without any previous knowledge of the correlation matrix of the input. In on-line learning the training patterns are not always defined in advance, and are generated one by one. A conservative approach is to minimize the risk by choosing a very small learning rate. In this case backpropagation can be trapped in a local minimum of a nonlinear error function. The learning rate should then be increased.

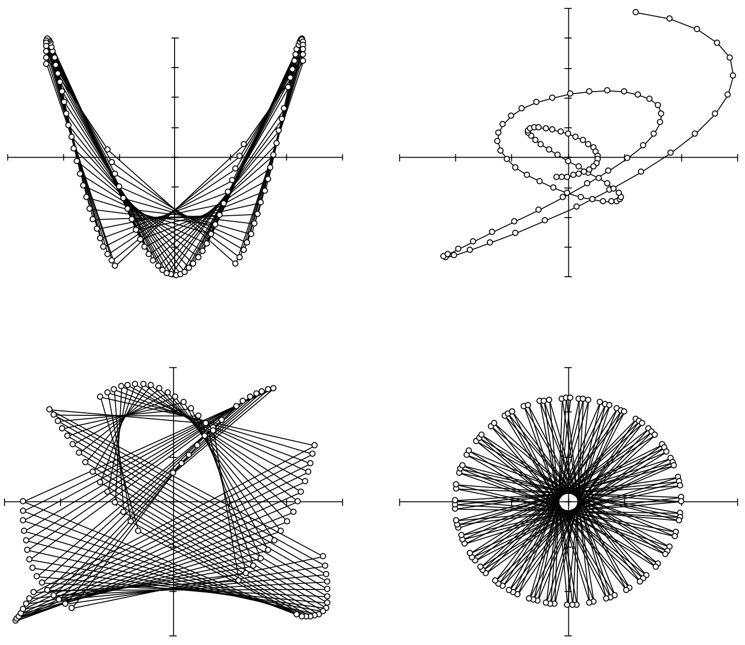


Fig. 8.5. Paths in weight space for backpropagation learning (linear associators). The minimum of the error function is located at the origin.

In the case of a correlation matrix $\mathbf{X}^{\mathrm{T}}\mathbf{X}$ with some very large eigenvalues, a given choice of γ could lead to divergence in the associated direction in weight space (assuming for simplicity that the principal directions of the quadratic form are aligned with the coordinate axis). Let us assume that the selected γ is near to the explosion point $2/k$ found in the one-dimensional case and shown in Figure 8.4. In this case only values of the momentum rate near to one can guarantee convergence, but oscillations in some of the directions in weight space can become synchronized. The result is oscillating paths in weight space, reminiscent of Lissajou's figures. Figure 8.5 shows some paths in a two-dimensional weight space for several linear associators trained with momentum rates close to one and different γ values. In some cases the trajectories lead to convergence after several thousand iterations. In others a momentum rate equal to one precludes convergence of the iteration process.

Adjustment of the learning and momentum rate in the nonlinear case is even more difficult than in the linear case, because there is no fast explosion of the iteration process. In the quadratic case, whenever the learning rate is excessively large, the iteration process leads rapidly to an overflow which alerts the programmer to the fact that the step size should be reduced. In the nonlinear case the output of the network and the error function are bounded and no overflow occurs. In regions far from local minima the gradient of the error function becomes almost zero as do the weight adjustments. The divergence regions of the quadratic case can now become oscillatory regions. In this case, even as step sizes increase, the iteration returns to the convex part of the error function.

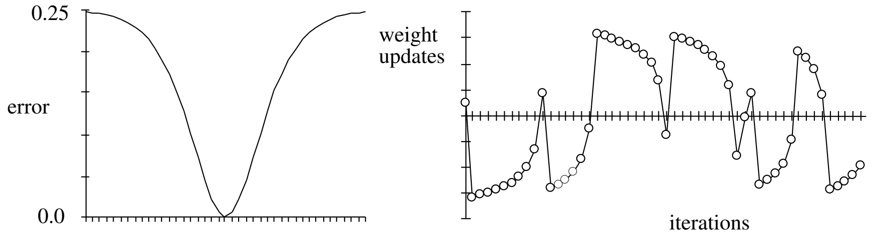


Fig. 8.6. Bounded nonlinear error function and the result of several iterations

Figure 8.6 shows the possible shape of the error function for a linear associator with sigmoidal output and the associated oscillation process for this kind of error function in the one-dimensional case. The jagged form of the iteration curve is reminiscent of the kind of learning curves shown in many papers about learning in nonlinear neural networks. Although in the quadratic case only large momentum rates lead to oscillations, in the nonlinear case an excessively large γ can also produce oscillations even when no momentum rate is present.

Researchers in the field of neural networks should be concerned not only with the possibility of getting stuck in local minima of the error function when learning rates are too small, but also with the possibility of falling into the oscillatory traps of backpropagation when the learning rate is too big. Learning algorithms should try to balance the speedup they are attempting to obtain with the risk of divergence involved in doing so. Two different kinds of remedy are available: a) adaptive learning rates and b) statistical preprocessing of the learning set which is done to decorrelate the input patterns, thus avoiding the negative effect of excessively large eigenvalues of the correlation matrix.

8.1.2 The fractal geometry of backpropagation

It is empirically known that standard backpropagation is very sensitive to the initial learning rate chosen for a given learning task. In this section we examine the shape of the iteration path for the training of a linear associator using on-line backpropagation. We show that the path is a fractal in weight space. The specific form depends on the learning rate chosen, but there is a threshold value for which the attractor of the iteration path is dense in a region of weight space around a local minimum of the error function. This result also yields a deeper insight into the mechanics of the iteration process in the nonlinear case.

The Gauss–Jacobi and Gauss–Seidel methods and backpropagation

Backpropagation can be performed in batch or on-line mode, that is, by updating the network weights once after each presentation of the complete training set or immediately after each pattern presentation. In general, on-line backpropagation does not converge to a single point in weight space, but oscillates around the minimum of the error function. The expected value of the deviation from the minimum depends on the size of the learning step being used. In this section we show that although the iteration process can fail to converge for some choices of learning rate, the iteration path for on-line learning is not just random noise, but possesses some structure and is indeed a fractal. This is rather surprising, because if the training patterns are selected randomly, one would expect a random iteration path. As we will see in what follows, it is easy to show that on-line backpropagation, in the case of linear associators, defines an *Iterated Function System* of the same type as the ones popularized by Barnsley [42]. This is sufficient proof that the iteration path has a fractal structure. To illustrate this fact we provide some computer-generated graphics.

First of all we need a visualization of the way off-line and on-line backpropagation approach the minimum of the error function. To simplify the discussion consider a linear associator with just two input lines (and thus two weights w_1 and w_2). Assume that three input-output patterns are given so that the equations to be satisfied are:

$$x_1^1 w_1 + x_2^1 w_2 = y_1 \tag{8.1}$$
$$x_1^2 w_1 + x_2^2 w_2 = y_2 \tag{8.2}$$
$$x_1^3 w_1 + x_2^3 w_2 = y_3 \tag{8.3}$$

These three equations define three lines in weight space and we look for the combination of w_1 and w_2 which satisfies all three simultaneously. If the three lines intersect at the same point, we can compute the solution using Gauss elimination. If the three lines do not intersect, no exact solution exists but we can ask for the combination of w_1 and w_2 which minimizes the quadratic error. This is the point inside the triangle shown in Figure 8.7, which has the minimal cumulative quadratic distance to the three sides of the triangle.

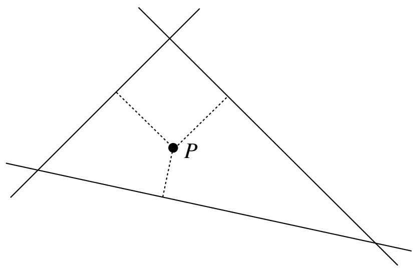


Fig. 8.7. Three linear constraints in weight space

Now for the interesting part: the point of intersection of linear equations can be found by linear algebraic methods such as the Gauss–Jacobi or the Gauss–Seidel method. Figure 8.8 shows how they work. If we are looking for the intersection of two lines, the Gauss–Jacobi method starts at some point in search space and projects this point in the directions of the axes on to the two lines considered in the example. The x-coordinate of the horizontal projection and the y-coordinate of the vertical projection define the next iteration point. It is not difficult to see that in the example this method converges to the point of intersection of the two lines. The Gauss–Seidel method deals with each linear equation individually. It first projects in the x direction, then in the y direction. Each projection is the new iteration point. This algorithm usually converges faster than the Gauss–Jacobi method [444].

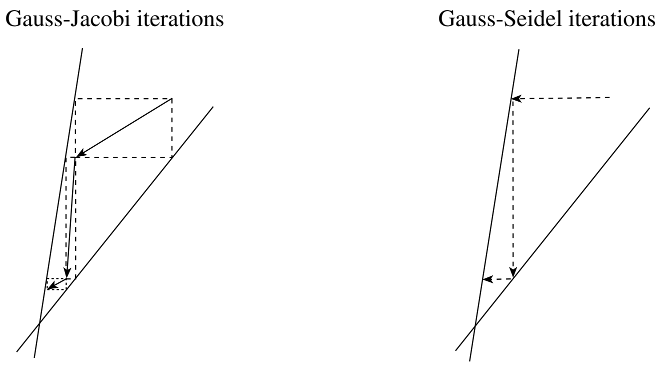


Fig. 8.8. The Gauss–Jacobi and Gauss–Seidel methods

Off-line and on-line backpropagation are in a certain sense equivalent to the Gauss–Jacobi and Gauss–Seidel methods. When on-line backpropagation is used, the negative gradient of the error function is followed and this corresponds to following a line perpendicular to each one of the equations (8.1) to (8.3). In the case of the first input-output pattern, the error function is

$$\frac{1}{2}(x_1^1 w_1 + x_2^1 w_2 - y_1)^2$$

and the gradient (with respect to w_1 and w_2) is the vector

$$(x_1, x_2)$$

which is normal to the line defined by equation (8.1). By randomly alternating the selection of each pattern, on-line backpropagation iterates, always moving in the direction normal to the linear constraints. Figure 8.9 shows that this method can be used to find the solution to linear equations. If the directions of the axis and the linear constraints coincide, on-line backpropagation is the Gauss–Seidel method with a learning constant.

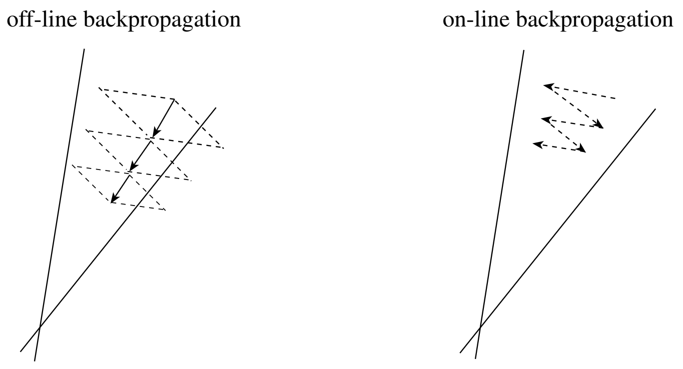


Fig. 8.9. Off-line and on-line backpropagation

Off-line backpropagation iterates by adding the corrections for each pattern. This means that the corrections in the direction normal to each linear constraint are calculated and the new iteration point is obtained by combining them. Figure 8.9 shows that this method is very similar to the Gauss–Jacobi method of linear algebra if we are looking for the solution to linear equations. Note that in both cases the size of the learning constant determines whether the iteration stops short of reaching the linear constraint or goes beyond it. This also depends on the curvature of the error function for each linear constraint (not shown in Figure 8.9).

In the nonlinear case, when a sigmoid is added as the activation function to the computing unit, the same kind of iteration process is used, but the sigmoid weights each one of the correction directions. On-line backpropagation always moves in the direction normal to the constraints, but the length of the search step is multiplied by the derivative of the sigmoid. Figure 8.10 shows the path of some iterations in the case of three input-output patterns and two weights.

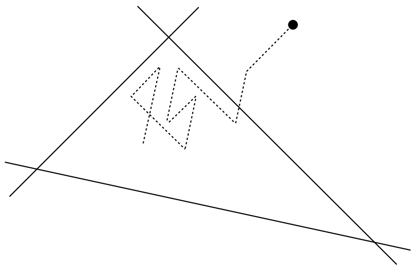


Fig. 8.10. On-line backpropagation iterations for sigmoidal units

Iterated Function Systems

Barnsley has shown that a set of affine transformations in a metric space can produce fractal structures when applied repetitively to a compact subset of points and its subsequent images. More specifically: an *Iterated Function System* (IFS) consists of a space of points X, a metric d defined in this space, and a set of affine contraction mappings $h_i : X \to X$, $i = 1, \ldots, N$. Given a nonvoid compact subset A_0 of points of X, new image subsets are computed successively according to the recursive formula

$$A_{n+1} = \bigcup_{j=1}^{N} h_j(A_n), \text{ for } n = 1, 2, \ldots.$$

A theorem guarantees that the sequence $\{A_n\}$ converges to a fixed point, which is called the *attractor* of the IFS. Moreover, the *Collage Theorem* guarantees that given any nonvoid compact subset of X we can always find an IFS whose associated attractor can arbitrarily approximate the given subset under a suitable metric.

For the case of an n-dimensional space X, an affine transformation applied to a point $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is defined by a matrix $\mathbf{M}$ and a vector $\mathbf{t}$. The transformation is given by

$$\mathbf{x} \to \mathbf{M}\mathbf{x} + \mathbf{t}.$$

and is contractive if the determinant of $\mathbf{M}$ is smaller than one.

It is easy to show that the attractor of the IFS can be approximated by taking any point a_o which belongs to the attractor and computing the sequence a_n, whereby

$$a_{n+1} = h_k(a_n),$$

and the affine transformation h_k is selected randomly from the IFS. The infinite sequence $\{a_n\}$ is a *dense* subset of the attractor. This means that we can produce a good graphical approximation of the attractor of the IFS with this simple randomized method. Since it is easy to approximate a point belonging to the attractor (by starting from the origin and iterating with the IFS a fixed number of times), this provides us with the necessary initial point.

We now proceed to show that on-line backpropagation, in the case of a linear associator, defines a set of affine transformations which are applied in the course of the learning process either randomly or in a fixed sequence. The iteration path of the initial point is thus a fractal.

On-line Backpropagation and IFS

Consider a linear associator and the p n-dimensional patterns $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^{\mathrm{p}}$. The symbol x_i^j denotes the i-th coordinate of the j-th pattern. The targets for training are the p real values $y^1, y^2, \ldots, y^p$. We denote by $w_1, w_2, \ldots, w_n$ the weights of the linear associator. In on-line backpropagation the error function is determined for each individual pattern. For pattern j the error is

$$E_j = \frac{1}{2}(w_1 x_1^j + w_2 x_2^j + \cdots + w_n x_n^j - y^j)^2.$$

The correction for each weight $w_i, i = 1, \ldots, n$, is given by

$$w_i \rightarrow w_i - \gamma x_i^j (w_1 x_1^j + w_2 x_2^j + \cdots + w_n x_n^j - y^j),$$

where γ is the learning constant being used. This can be also written as

$$w_i \rightarrow w_i - \gamma (w_1 x_i^j x_1^j + w_2 x_i^j x_2^j + \cdots + w_n x_i^j x_n^j) - \gamma x_i^j y^j \,. \qquad (8.4)$$

Let M_j be the matrix with elements $m_{ik} = x_i^j x_k^j$ for $i, k = 1, \ldots, n$. Equation (8.4) can be rewritten in matrix form (considering all values of i) as

$$\mathbf{w} \rightarrow \mathbf{I}\mathbf{w} - \gamma \mathbf{M}_j \mathbf{w} - \mathbf{t}_j,$$

where $\mathbf{t}_j$ is the column vector with components $x_i^j y^j$ for $i = 1, \ldots, n$ and $\mathbf{I}$ is the identity matrix. We can rewrite the above expression as

$$\mathbf{w} \rightarrow (\mathbf{I} - \gamma \mathbf{M}_j)\mathbf{w} - \mathbf{t}_j.$$

This is an affine transformation, which maps the current point in weight space into a new one. Note that each pattern in the training set defines a different affine transformation of the current point $(w_1, w_2, \ldots, w_n)$.

In on-line backpropagation with a random selection of input patterns, the initial point in weight space is successively transformed in just the way prescribed by a randomized IFS algorithm. This means that the sequence of updated weights approximates the attractor of the IFS defined by the input patterns, i.e., the iteration path in weight space is a fractal.

This result can be visualized in the following manner: given a point $(w_1', w_2', \ldots, w_n')$ in weight space, the square of the distance ℓ to the hyperplane defined by the equation

$$w_1 x_1^j + w_2 x_2^j + \cdots + w_n x_n^j = y^j$$

Fig. 8.11. Iteration paths in weight space for different learning rates

is given by

$$\ell^2 = \frac{(w_1' x_1^j + w_2' x_2^j + \cdots + w_n' x_n^j - y^j)^2}{(x_1^j)^2 + \cdots + (x_n^j)^2}.$$

Comparing this expression with the updating step performed by on-line backpropagation, we see that each backpropagation step amounts to displacing the current point in weight space in the direction normal to the hyperplane defined by the input and target patterns. A learning rate γ with value

$$\gamma = \frac{1}{(x_1^j)^2 + \cdots + (x_n^j)^2} \tag{8.5}$$

produces exact learning of the j-th pattern, that is, the point in weight space is brought up to the hyperplane defined by the pattern. Any value of γ below this threshold displaces the iteration path just a fraction of the distance to the hyperplane. The iteration path thus remains trapped in a certain region of weight space near to the optimum.

Figure 8.11 shows a simple example in which three input-output patterns define three lines in a two-dimensional plane. Backpropagation for a linear associator will find the point in the middle as the solution with the minimal error for this task. Some combinations of learning rate, however, keep the iteration path a significant distance away from this point. The fractal structure of the iteration path is visible in the first three graphics. With $\gamma = 0.25$ the fractal structure is obliterated by a dense approximation to the whole triangular region. For values of γ under 0.25 the iteration path repeatedly comes arbitrarily near to the local minimum of the error function.

It is clear that on-line backpropagation with random selection of the input patterns yields iteration dynamics equivalent to those of IFS. When the learning constant is such that the affine transformations overlap, the iteration path densely covers a certain region of weight space around the local minimum of the error function. Practitioners know that they have to systematically reduce the size of the learning step, because otherwise the error function begins oscillating. In the case of linear associators and on-line backpropagation this means that the iteration path has gone fractal. Chaotic behavior of recurrent neural networks has been observed before [281], but in our case we are dealing with very simple feed-forward networks which fall into a similar dynamic.

In the case of sigmoid units at the output, the error correction step is no longer equivalent to an affine transformation, but the updating step is very similar. It can be shown that the error function for sigmoid units can approximate a quadratic function for suitable parameter combinations. In this case the iteration dynamics will not differ appreciably from those discussed in this chapter. Nonlinear updating steps should also produce fractal iteration paths of a more complex nature.

8.2 Some simple improvements to backpropagation

Since learning in neural networks is an *NP*-complete problem and since traditional gradient descent methods are rather slow, many alternatives have been tried in order to accelerate convergence. Some of the proposed methods are mutually compatible and a combination of them normally works better than each method alone [340]. But apart from the learning algorithm, the basic point to consider before training a neural network is *where* to start the iterative learning process. This has led to an analysis of the best possible weight initialization strategies and their effect on the convergence speed of learning [328].

8.2.1 Initial weight selection

A well-known initialization heuristic for a feed-forward network with sigmoidal units is to select its weights with uniform probability from an interval $[-\alpha, \alpha]$. The zero mean of the weights leads to an expected zero value of the total input to each node in the network. Since the derivative of the sigmoid at each node reaches its maximum value of 1/4 with exactly this input, this should lead in principle to a larger backpropagated error and to more significant weight updates when training starts. However, if the weights in the networks are very small (or all zero) the backpropagated error from the output to the hidden layer is also very small (or zero) and practically no weight adjustment takes place between input and hidden layer. Very small values of α paralyze learning, whereas very large values can lead to saturation of the nodes in the network and to flat zones of the error function in which, again, learning is very slow. Learning then stops at a suboptimal local minimum [277]. Therefore it is natural to ask what is the best range of values for α in terms of the learning speed of the network.

Some authors have conducted empirical comparisons of the values for α and have found a range in which convergence is best [445]. The main problem with these results is that they were obtained from a limited set of examples and the relation between learning step and weight initialization was not considered. Others have studied the percentage of nodes in a network which become paralyzed during training and have sought to minimize it with the "best" α [113]. Their empirical results show, nevertheless, that there is not a single α which works best, but a very broad range of values with basically the same convergence efficiency.

Maximizing the derivatives at the nodes

Let us first consider the case of an output node. If n different edges with associated weights $w_1, w_2, \ldots, w_n$ point to this node, then after selecting weights with uniform probability from the interval $[-\alpha, \alpha]$, the expected total input to the node is

$$\left\langle \sum_{i=1}^{n} w_i x_i \right\rangle = 0$$

where $x_1, x_2, \ldots, x_n$ are the input values transported through each edge. We have assumed that these inputs and the initial weights are not correlated. By the law of large numbers we can also assume that the total input to the node has a Gaussian distribution Numerical integration shows that the expected value of the derivative is a decreasing function of the standard deviation σ. The expected value falls slowly with an increase of the variance. For $\sigma = 0$ the expected value is 0.25 and for $\sigma = 4$ it is still 0.12, that is, almost half as big.

The variance of the total input to a node is

$$\sigma^2 = E((\sum_{i=1}^{n} w_i x_i)^2) - E((\sum_{i=1}^{n} w_i x_i))^2 = \sum_{i=1}^{n} E(w_i^2)E(x_i^2),$$

since inputs and weights are uncorrelated. For binary vectors we have $E(x_i^2) = 1/3$ and the above equation simplifies to

$$\sigma = \frac{1}{3}\sqrt{n}\,\alpha.$$

If $n = 100$, selecting weights randomly from the interval $[-1.2, 1.2]$ leads to a variance of 4 at the input of a node with 100 connections and to an expected value of the derivative equal to 0.12.

Therefore in small networks, in which the maximum input to each node comes from fewer than 100 edges, the expected value of the derivative is not very sensitive to the width α of the random interval, when α is small enough.

Maximizing the backpropagated error

In order to make corrections to the weights in the first block of weights (those between input and hidden layer) easier, the backpropagated error should be as large as possible. Very small weights between hidden and output nodes lead to a very small backpropagated error, and this in turn to insufficient corrections to the weights. In a network with m nodes in the hidden layer and k nodes in the output layer, each hidden node h receives a backpropagated input δ_h from the k output nodes, equal to

$$\delta_h = \sum_{i=1}^{k} w_{hi} s'_i \delta_i^0,$$

where the weights $w_{hi}, i = 1, \ldots, k$, are the ones associated with the edges from hidden node h to output node i, s'_i is the sigmoid's derivative at the output node i, and δ_i^0 is the difference between output and target also at this node.

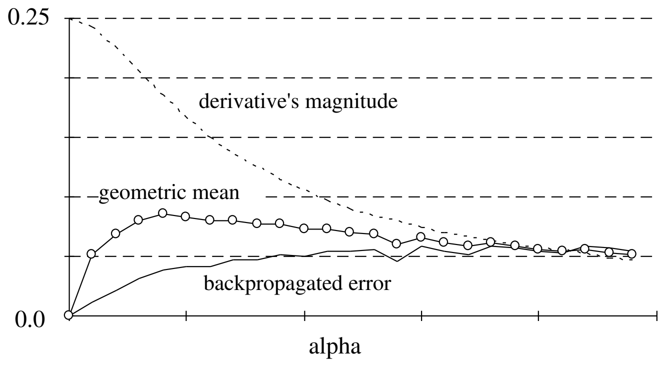


Fig. 8.12. Expected values of the backpropagated error and the sigmoid's derivative

After initialization of the network's weights the expected value of δ_h is zero. In the first phase of learning we are interested in *breaking the symmetry* of the hidden nodes. They should specialize in the recognition of different features of the input. By making the variance of the backpropagated error larger, each hidden node gets a greater chance of pulling apart from its neighbors. The above equation shows that by making the initialization interval $[-\alpha, \alpha]$ wider, two contradictory forces come into play. On the one hand, the variance of the weights becomes larger, but on the other hand, the expected value of the derivative s'_k becomes lower. We would like to make δ_h as large as possible, but without making s'_i too low, since weight corrections in the second block of weights are proportional to s'_i. Figure 8.12 shows the expected values of the derivative at the output nodes, the expected value of the backpropagated error for the hidden nodes as a function of α, and the geometric mean of both values. The data in the figure was obtained from Monte Carlo trials, assuming a constant expected value of δ_i^0. Forty hidden and one output unit were used.

The figure shows, again, that the expected value of the sigmoid's derivative falls slowly with an increasing α, but the value of the backpropagated error is sensitive to small values of α. In the case shown in the figure, any possible choice of α between 0.5 and 1.5 should lead to virtually the same performance. This explains the flat region of possible values for α found in the experiments published in [113] and [445]. Consequently, the best values for α depend on the exact number of input, hidden, and output units, but the learning algorithm should not be very sensitive to the exact α chosen from a certain range of values.

8.2.2 Clipped derivatives and offset term

One of the factors which leads to slow convergence of gradient descent methods is the small absolute value of the partial derivatives of the error function computed by backpropagation. The derivatives of the sigmoid stored at the computing units can easily approach values near to zero and since several of them can be multiplied in the backpropagation step, the length of the gradient can become too small. A solution to this problem is clipping the derivative of the sigmoid, so that it can never become smaller than a predefined value. We could demand, for example, that $s'(x) \geq 0.01$. In this case the "derivatives" stored in the computing units do not correspond exactly to the actual derivative of the sigmoid (except in the regions where the derivative is not too small). However, the partial derivatives have the correct sign and the gradient direction is not significantly affected.

It is also possible to add a small constant ε to the derivative and use $s'(x)+\varepsilon$ for the backpropagation step. The net effect of an offset value for the sigmoid's derivative is to pull the iteration process out of flat regions of the error function. Once this has happened, backpropagation continues iterating with the exact gradient direction. It has been shown in many different learning experiments that this kind of heuristic, proposed by Fahlman [130], among others, can contribute significantly to accelerate several different variants of the standard backpropagation method [340]. Note that the offset term can be implemented very easily when the sigmoid is not computed at the nodes but only read from a table of function values. The table of derivatives can combine clipping of the sigmoid values with the addition of an offset term, to enhance the values used in the backpropagation step.

8.2.3 Reducing the number of floating-point operations

Backpropagation is an expensive algorithm because a straightforward implementation is based on floating-point operations. Since all values between 0 and 1 are used, problems of precision and stability arise which are normally avoided by using 32-bit or 64-bit floating-point arithmetic. There are several possibilities to reduce the number of floating-point operations needed.

Avoiding the computation of the squashing function

If the nonlinear function used at each unit is a sigmoid or the hyperbolic tangent, then an exponential function has to be computed and this requires a sequence of floating-point operations. However, computation of the nonlinearity can be avoided by using tables stored at each unit, in which for an interval $[x_i, x_{i+1}]$ in the real line the corresponding approximation to the sigmoid is stored. A piecewise linear approximation can be used as shown in Figure 8.13, so that the output of the unit is $y = a_i + a_{i+1}(x - x_i)$ when $x_i \leq x < x_{i+1}$ and where $a_1 = s(x_i)$ and $a_{i+1} = s(x_{i+1})$. Computation of the nonlinear function is reduced in this way to a comparison, a table lookup, and an interpolation. Another table holding some values of the sigmoid's derivative can be stored at each unit for the backpropagation step. A piecewise linear approximation can also be used in this case. This strategy is used in chips for neural computation in order to avoid using many logic gates.

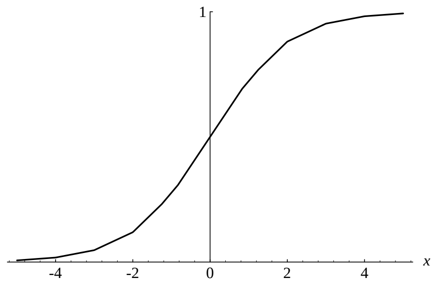


Fig. 8.13. Piecewise linear approximation to the sigmoid

Avoiding the nonlinear function at the output

In some cases the sigmoid at the output of the network can be eliminated. If the output vectors in the training set are m-dimensional vectors of the form $(t_1, \ldots, t_m)$ with $0 < t_i < 1$ for $i = 1, \ldots, m$, then a new training set can be defined with the same input vectors and output vectors of the form $(s^{-1}(y_1), \ldots, s^{-1}(y_m))$, where the function s^{-1} is the inverse of the sigmoid. The sigmoid is eliminated from the output units and the network is trained with standard backpropagation. This strategy will save some operations but its applicability depends on the problem, since the sigmoid is equivalent to a kind of weighting of the output error. The inputs 100 or 1000 produce almost the same sigmoid output, but the two numbers are very different when compared directly. Only more knowledge about the specific application can help to decide if the nonlinearity at the output can be eliminated (see Sect. 9.1.3 on logistic regression).

Fixed-point arithmetic

Since integer operations can be executed in many processors faster than floating-point operations, and since the outputs of the computing units are values in the interval $(0,1)$ or $(-1,1)$, it is possible to adopt a fixed-point representation and perform all necessary operations with integers. By convention we can define the last 12 bits of a number as its fractional part and the three previous bits as its integer part. Using a sign bit it is possible to represent numbers in the interval $(-8,8)$ with a precision of $2^{-12} \approx 0.00025$. Care has to be taken to re-encode the input and output vectors, to define the tables for the sigmoid and its derivatives and to implement the correct arithmetic (which requires a shift after multiplications and tests to avoid overflow). Most of the more popular neural chips implement some form of fixed-point arithmetic (see Chap. 16).

Some experiments show that in many applications it is enough to reserve 16 bits for the weights and 8 for the coding of the outputs, without affecting the convergence of the learning algorithm [31]. Holt and Baker compared the results produced by networks with floating-point and fixed-point parameters using the Carnegie-Mellon benchmarks [197]. Their results confirmed that a combination of 16-bit and 8-bit coding produces good results. In four of five benchmarks the result of the comparison was "excellent" for fixed-point arithmetic and in the other case "good". Based on these results, groups developing neural computers like the CNAPS of Adaptive Solutions [175] and SPERT in Berkeley [32] decided to stick to 16-bit and 8-bit representations.

Reyneri and Filippi [364] did more extensive experimentation on this problem and arrived at the conclusion that the necessary word length of the representation depends on the learning constant and *the kind of learning algorithm used*. This was essentially confirmed by the experiments done by Pfister on a fixed-point neurocomputer [341]. Standard backpropagation can diverge in some cases when the fixed-point representation includes less than 16 bits. However, modifying the learning algorithm and adapting the learning constant reduced the necessary word length to 14 bits. With the modified algorithm 16 bits were more than enough.

8.2.4 Data decorrelation

We have already mentioned that if the principal axes of the quadratic approximation of the error function are too dissimilar, gradient descent can be slowed down arbitrarily. The solution lies in decorrelating the data set and there is now ample evidence that this preconditioning step is beneficial for the learning algorithm [404, 340].

One simple decorrelation strategy consists in using bipolar vectors. We showed in Chap. 6 that the solution regions defined by bipolar data for perceptrons are more symmetrically distributed than when binary vectors are used. The same holds for multilayer networks. Pfister showed that convergence of the standard backpropagation algorithm can be improved and that a speedup between 1.91 and 3.53 can be achieved when training networks for several small benchmarks (parity and clustering problems) [341]. The exception to this general result are encode-decode problems in which the data consists of sparse vectors (n-dimensional vectors with a single 1 in a component). In this case binary coding helps to focus on the corrections needed for the relevant weights. But if the data consists of non-sparse vectors, bipolar coding usually works better. If the input data consists of N real vectors $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N$ it is usually helpful to center the data around the origin by subtracting the centroid $\hat{\mathbf{x}}$ of the distribution, in such a way that the new input data consists of the vectors $\mathbf{x}_i - \hat{\mathbf{x}}$.

PCA and adaptive decorrelation

Centering the input and output data is just the first step in more sophisticated preconditioning methods. One of them is principal component analysis, already discussed in Chap. 5. Using PCA it is possible to reduce the number of vector components (when there is redundancy) and to order the remaining ones according to their importance.

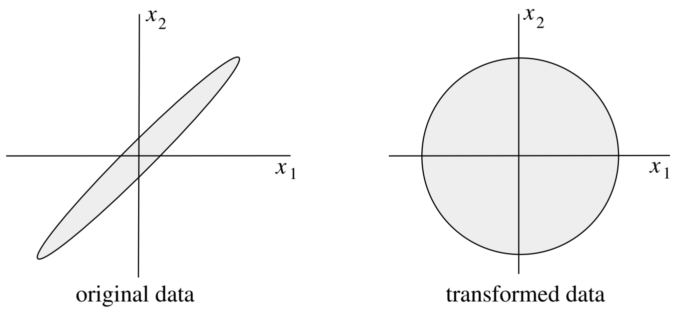


Fig. 8.14. Data distribution before and after preconditioning

Assume that the data distribution is the one shown on the left side of Figure 8.14. Any of the points in the ellipsoid can be a future input vector to the network. Two vectors selected randomly from the distribution have a large probability of not being orthogonal (since their components are correlated). After a rotation of the axes and a scaling step the data distribution becomes the one shown to the right in Figure 8.14. Note that the transformation is one-to-one and therefore invertible. But now, any two vectors selected randomly from the new distribution (a sphere in n-dimensional space) have a greater probability of being orthogonal. The transformation of the data is linear (a rotation followed by scaling) and can be applied to new data as it is provided.

Note that we do not know the exact form of the data distribution. All we have is a training set (the dots in Figure 8.14) from which we can compute the optimal transformation. Principal component analysis is performed on the available data. This gives us a new encoding for the data set and also a linear transformation for new data. The scaling factors are the reciprocal of the variances of each component (see Exercise 2). PCA preconditioning speeds up backpropagation in most cases, except when the data consists of sparse vectors.

Silva and Almeida have proposed another data decorrelation technique, called Adaptive Data Decorrelation, which they use to precondition data [404]. A linear layer is used to transform the input vectors, and another to transform the outputs of the hidden units. The linear layer consists of as many output as input units, that is, it only applies a linear transformation to the vectors. Consider an n-dimensional input vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$. Denote the output of the linear layer by $(y_1, y_2, \ldots, y_n)$. The objective is to make the expected value of the correlation coefficient r_{ij} of the i-th and j-th output units equal to Kronecker's delta δ_{ij}, i.e.,

$$r_{ij} =< y_i y_j >= \delta_{ij}.$$

The expected value is computed over all vectors in the data set. The algorithm proposed by Silva and Almeida is very simple: it consists in pulling the weight vector of the i-th unit away from the weight vector of the j-th unit, whenever $r_{ij} > 0$ and in the direction of w_j when $r_{ij} < 0$. The precise formula is

$$\mathbf{w}_i^{k+1} = \mathbf{w}_i^k - \beta \sum_{j \neq i}^{n} r_{ij} \mathbf{w}_j^k,$$

where $\mathbf{w}_m^p$ denotes the weight vector of the m-th unit at the p-th iteration and β is a constant. Data normalization is achieved by including a positive or negative term according to whether r_{ii} is smaller or greater than 1:

$$\mathbf{w}_i^{k+1} = (1 + \beta)\mathbf{w}_i^k - \beta \sum_{j=1}^{n} r_{ij} \mathbf{w}_j^k.$$

The proof that the algorithm converges (under certain assumptions) can be found in [404]. Adaptive Decorrelation can accelerate the convergence of backpropagation almost as well as principal component analysis, but is somewhat sensitive to the choice of the constant β and the data selection process [341].

Active data selection

For large training sets, redundancy in the data can make prohibitive the use of off-line backpropagation techniques. On-line backpropagation can still perform well under these conditions if the training pairs are selected randomly. Since some of the fastest variations of backpropagation work off-line, there is a strong motivation to explore methods of reducing the size of the effective training set.

Some authors have proposed training a network with a subset of the training set, adding iteratively the rest of the training pairs [261]. This can be done by testing the untrained pairs. If the error exceeds a certain threshold (for example if it is one standard deviation larger than the average error on the effective training set), the pair is added to the effective training set and the network is retrained when more than a certain number of pairs have been recruited [369].

8.3 Adaptive step algorithms

The class of adaptive step algorithms uses a very similar basic strategy: the step size is increased whenever the algorithm proceeds down the error function over several iterations. The step size is decreased when the algorithm jumps over a valley of the error function. The algorithms differ according to the kind of information used to modify the step size.

In learning algorithms with a *global* learning rate, this is used to update all weights in the network. The learning rate is made bigger or smaller according to the iterations made so far.

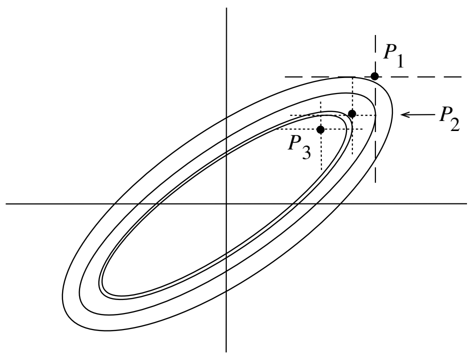


Fig. 8.15. Optimization of the error function with updates in the directions of the axes

In algorithms with a *local* learning constant a different constant is used for each weight. Whereas in standard backpropagation a single constant γ is used to compute the weight corrections, in four of the algorithms considered below each weight w_i has an associated learning constant γ_i so that the updates are given by

$$\Delta w_i = -\gamma_i \frac{\partial E}{\partial w_i}.$$

The motivation behind the use of different learning constants for each weight is to try to "correct" the direction of the negative gradient to make it point directly to the minimum of the error function. In the case of a degenerate error function, the gradient direction can lead to many oscillations. An adequate scaling of the gradient components can help to reach the minimum in fewer steps.

Figure 8.15 shows how the optimization of the error function proceeds when only one-dimensional optimization steps are used (from point P_1 to P_2, and then to P_3). If the lengths of the principal axes of the quadratic approximation are very different, the algorithm can be slowed by an arbitrary factor.

8.3.1 Silva and Almeida's algorithm

The method proposed by Silva and Almeida works with different learning rates for each weight in a network [403]. Assume that the network consists of n weights $w_1, w_2, \ldots, w_n$ and denote the individual learning rates by $\gamma_1, \gamma_2, \ldots, \gamma_n$. We can better understand how the algorithm works by looking at Figure 8.16. The left side of the figure shows the level curves of a quadratic approximation to the error function. Starting the iteration process as described before and as illustrated with Figure 8.15, we can see that the algorithm performs several optimization steps in the horizontal direction. Since horizontal cuts to a quadratic function are quadratic, what we are trying to minimize at each step is one of the several parabolas shown on the right side of Figure 8.16. A quadratic function of n variable weights has the general form

$$c_1^2 w_1^2 + c_2^2 w_2^2 + \cdots + c_n^2 w_n^2 + \sum_{i \neq j} d_{ij} w_i w_j + C,$$

where $c_1, \ldots, c_n$, the d_{ij} and C are constants. If the i-th direction is chosen for a one-dimensional minimization step the function to be minimized has the form

$$c_i^2 w_i^2 + k_1 w_i + k_2,$$

where k_1 and k_2 are constants which depend on the values of the 'frozen' variables at the current iteration point. This equation defines a family of parabolas. Since the curvature of each parabola is given by c_i^2, they differ just by a translation in the plane, as shown in Figure 8.16. Consequently, it makes sense to try to find the optimal learning rate for this direction, which is equal to $1/2c_i^2$, as discussed in Sect. 8.1.1. If we have a different learning rate, optimization proceeds as shown on the left of Figure 8.16: the first parabola is considered and the negative gradient direction is followed. The iteration steps in the other dimensions change the family of parabolas which we have to consider in the next step, but in this case the negative gradient direction is also followed. The family of parabolas changes again and so on. With this quadratic approximation in mind the question then becomes, what are the optimal values for γ_1 to γ_n? We arrive at an additional optimization problem!

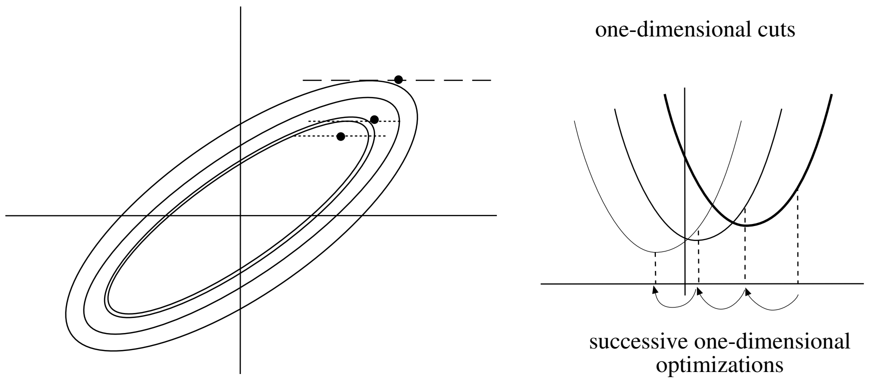


Fig. 8.16. One-dimensional cuts: family of parabolas

The heuristic proposed by Almeida is very simple: accelerate if, in two successive iterations, the sign of the partial derivative has not changed, and decelerate if the sign changes. Let $\nabla_i E^{(k)}$ denote the partial derivative of the error function with respect to weight w_i at the k-th iteration. The initial learning rates $\gamma_i^{(0)}$ for $i = 1, \ldots, n$ are initialized to a small positive value. At the k-th iteration the value of the learning constant for the next step is recomputed for each weight according to

$$\gamma_i^{(k+1)} = \begin{cases} \gamma_i^{(k)} u \text{ if } \nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} \geq 0 \\ \gamma_i^{(k)} d \text{ if } \nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} < 0 \end{cases}$$

The constants u (up) and d (down) are set by hand with $u > 1$ and $d < 1$. The weight updates are made according to

$$\Delta^{(k)} w_i = -\gamma_i^{(k)} \nabla_i E^{(k)} .$$

Note that due to the constants u and d the learning rates grow and decrease *exponentially*. This can become a problem if too many acceleration steps are performed successively. Obviously this algorithm does not follow the gradient direction exactly. If the level curves of the quadratic approximation of the error function are perfect circles, successive one-dimensional optimizations lead to the minimum after n steps. If the quadratic approximation has semi-axes of very different lengths, the iteration process can be arbitrarily slowed. To avoid this, the updates can include a momentum term with rate α. Nevertheless, both kinds of corrections together are somewhat contradictory: the individual learning rates can only be optimized if the optimization updates are strictly one-dimensional. Tuning the constant α can therefore become quite problem-specific. The alternative is to preprocess the original data to achieve a more regular error function. This can have a dramatic effect on the convergence speed of algorithms which perform one-dimensional optimization steps.

8.3.2 Delta-bar-delta

The algorithm proposed by Jacobs is similar to Silva and Almeida's, the main difference being that acceleration of the learning rates is made with more caution than deceleration. The algorithm is started with individual learning rates $\gamma_1, \ldots, \gamma_n$ all set to a small value, and at the k-th iteration the new learning rates are set to

$$\gamma_i^{(k+1)} = \begin{cases} \gamma_i^{(k)} + u & \text{if } \nabla_i E^{(k)} \cdot \delta_i^{(k-1)} > 0 \\ \gamma_i^{(k)} d & \text{if } \nabla_i E^{(k)} \cdot \delta_i^{(k-1)} < 0 \\ \gamma_i^{(k)} & \text{otherwise,} \end{cases}$$

where u and d are constants and $\delta_i^{(k)}$ is an exponentially averaged partial derivative in the direction of weight w_i:

$$\delta_i^{(k)} = (1-\phi)\nabla_i E^{(k)} + \phi \delta_i^{(k-1)}.$$

The constant ϕ determines what weight is given to the last averaged term. The weight updates are performed without a momentum term:

$$\Delta^{(k)} w_i = -\gamma_i^{(k)} \nabla_i E^{(k)}.$$

The motivation for using an averaged gradient is to avoid excessive oscillations of the basic algorithm. The problem with this approach, however, is that a new constant has to be set again by the user and its value can also be highly problem-dependent. If the error function has regions which allow a good quadratic approximation, then $\phi = 0$ is optimal and we are essentially back to Silva and Almeida's algorithm.

8.3.3 Rprop

A variant of Silva and Almeida's algorithm is Rprop, first proposed by Riedmiller and Braun [366]. The main idea of the algorithm is to update the network weights using just the learning rate and the sign of the partial derivative of the error function with respect to each weight. This accelerates learning mainly in flat regions of the error function as well as when the iteration has arrived close to a local minimum. To avoid accelerating or decelerating too much, a minimum value for the learning rates γ_{min} and a maximum value γ_{max} is enforced. The algorithm covers all of weight space with an n-dimensional grid of side γ_{min} and an n-dimensional grid of side length γ_{max}. The individual one-dimensional optimization steps can move on all possible intermediate grids. The learning rates are updated in the k-th iteration according to

$$\gamma_i^{(k+1)} = \begin{cases} \min(\gamma_i^{(k)}u, \gamma_{max}) \text{ if } \nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} > 0 \\ \max(\gamma_i^{(k)}d, \gamma_{min}) \text{ if } \nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} < 0 \\ \gamma_i^{(k)} \qquad \text{otherwise,} \end{cases}$$

where the constants u and d satisfy $u > 1$ and $d < 1$, as usual. When $\nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} \geq 0$ the weight updates are given by

$$\Delta^{(k)} w_i = -\gamma_i^{(k)} \operatorname{sgn}(\nabla_i E^{(k)}),$$

otherwise $\Delta^{(k)} w_i$ and $\nabla_i E^{(k)}$ are set to zero. In the above equation $\operatorname{sgn}(\cdot)$ denotes the sign function with the peculiarity that $\operatorname{sgn}(0) = 0$.

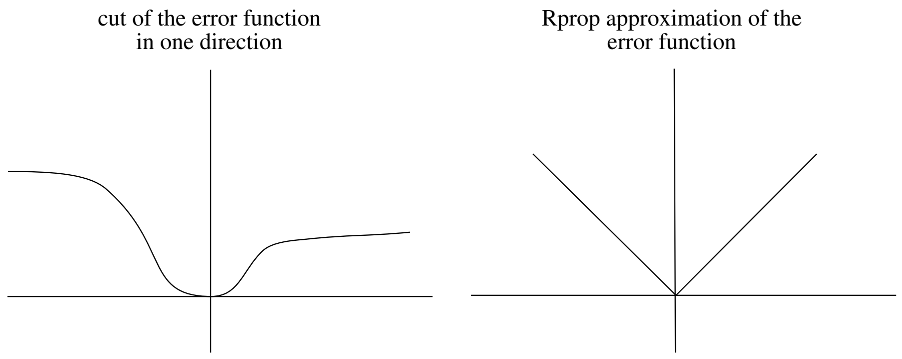

cut of the error function in one direction
Rprop approximation of the error function

Fig. 8.17. Local approximation of Rprop

Figure 8.17 shows the kind of one-dimensional approximation of the error function used by Rprop. It may seem a very imprecise approach, but it works very well in the flat regions of the error function. Schiffmann, Joost, and Werner tested several algorithms using a medical data set and Rprop produced the best results, being surpassed only by the constructive algorithm called *cascade correlation* [391] (see Chap. 14).

Table 8.1 shows the results obtained by Pfister on some of the Carnegie Mellon Benchmarks [341]. The training was done on a CNAPS neurocomputer. Backpropagation was coded using some optimizations (bipolar vectors, offset term, etc.). The table shows that BP did well for most benchmarks but failed to converge in two of them. Rprop converged almost always (98% of

Table 8.1. Comparison of Rprop and batch backpropagation

benchmark	generations		time	
	BP	Rprop	BP	Rprop
sonar signals	109.7	82.0	8.6 s	6.9 s
vowels	–	1532.9	–	593.6 s
vowels (decorrelated)	331.4	319.1	127.8 s	123.6 s
NETtalk (200 words)	268.9	108.7	961.5 s	389.6 s
protein structure	347.5	139.2	670.1 s	269.1 s
digits	–	159.5	–	344.7 s

the time) and was faster by up to a factor of about 2.5 with respect to batch backpropagation.

It should be pointed out that the vowels recognition task was presented in two versions, one without and one with decorrelated inputs. In the latter case, backpropagation did converge and was almost as efficient as Rprop. This shows how important the preprocessing of the input data can be. Note that the overall speedup obtained is limited because the version of backpropagation used for the comparison was rather efficient.

8.3.4 The Dynamic Adaption algorithm

We close our discussion of adaptive step methods with an algorithm based on a global learning rate [386]. The idea of the method is to use the negative gradient direction to generate two new points instead of one. The point with the lowest error is used for the next iteration. If it is the farthest away the algorithm accelerates, by making the learning constant bigger. If it is the nearest one, the learning constant γ is reduced.

The k-th iteration of the algorithm consists of the following three steps:

- Compute

$$\begin{aligned}\mathbf{w}^{(k_1)} &= \mathbf{w}^{(k)} - \nabla E(\mathbf{w}^{(k)})\gamma^{(k-1)} \cdot \xi \\ \mathbf{w}^{(k_2)} &= \mathbf{w}^{(k)} - \nabla E(\mathbf{w}^{(k)})\gamma^{(k-1)}/\xi\end{aligned}$$

  where ξ is a small constant (for example $\xi = 1.7$).
- Update the learning rate:

$$\gamma^{(k)} = \begin{cases} \gamma^{(k-1)} \cdot \xi & \text{if } E(\mathbf{w}^{(k_1)}) \leq E(\mathbf{w}^{(k_2)}) \\ \gamma^{(k-1)}/\xi & \text{otherwise.} \end{cases}$$

- Update the weights:

$$\mathbf{w}^{(k+1)} = \begin{cases} \mathbf{w}^{(k_1)} & \text{if } E(\mathbf{w}^{(k_1)}) \leq E(\mathbf{w}^{(k_2)}) \\ \mathbf{w}^{(k_2)} & \text{otherwise.} \end{cases}$$

The algorithm is not as good as the adaptive step methods with a local learning constant, but is very easy to implement. The overhead involved is an extra feed-forward step.

8.4 Second-order algorithms

The family of second-order algorithms considers more information about the shape of the error function than the mere value of the gradient. A better iteration can be performed if the curvature of the error function is also considered at each step. In second-order methods a quadratic approximation of the error function is used [43]. Denote all weights of a network by the vector $\mathbf{w}$. Denote the error function by $E(\mathbf{w})$. The truncated Taylor series which approximates the error function E is given by

$$E(\mathbf{w}+\mathbf{h}) \approx E(\mathbf{w}) + \nabla E(\mathbf{w})^{\mathrm{T}}\mathbf{h} + \frac{1}{2}\mathbf{h}^{\mathrm{T}}\nabla^2 E(\mathbf{w})\mathbf{h}, \tag{8.6}$$

where $\nabla^2 E(\mathbf{w})$ is the $n \times n$ Hessian matrix of second-order partial derivatives:

$$\nabla^2 E(\mathbf{w}) = \begin{pmatrix} \frac{\partial^2 E(\mathbf{w})}{\partial w_1^2} & \frac{\partial^2 E(\mathbf{w})}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 E(\mathbf{w})}{\partial w_1 \partial w_n} \\ \frac{\partial^2 E(\mathbf{w})}{\partial w_2 \partial w_1} & \frac{\partial^2 E(\mathbf{w})}{\partial w_2^2} & \cdots & \frac{\partial^2 E(\mathbf{w})}{\partial w_2 \partial w_n} \\ \vdots & & \ddots & \vdots \\ \frac{\partial^2 E(\mathbf{w})}{\partial w_n \partial w_1} & \frac{\partial^2 E(\mathbf{w})}{\partial w_n \partial w_2} & \cdots & \frac{\partial^2 E(\mathbf{w})}{\partial w_n^2} \end{pmatrix}.$$

The gradient of the error function can be computed by differentiating (8.6):

$$\nabla E(\mathbf{w}+\mathbf{h})^{\mathrm{T}} \approx \nabla E(\mathbf{w})^{\mathrm{T}} + \mathbf{h}^{\mathrm{T}}\nabla^2 E(\mathbf{w}).$$

Equating to zero (since we are looking for the minimum of E) and solving, we get

$$\mathbf{h} = -(\nabla^2 E(\mathbf{w}))^{-1}\nabla E(\mathbf{w}), \tag{8.7}$$

that is, the minimization problem can be solved in a single step if we have previously computed the Hessian matrix and the gradient, of course under the assumption of a quadratic error function.

Newton's method works by using equation (8.7) iteratively. If we denote now the weight vector at the k-th iteration by $\mathbf{w}^{(k)}$, the new weight vector $\mathbf{w}^{(k+1)}$ is given by

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - (\nabla^2 E(\mathbf{w}))^{-1}\nabla E(\mathbf{w}). \tag{8.8}$$

Under the quadratic approximation, this will be a position where the gradient has reduced its magnitude. Iterating several times we can get to the minimum of the error function.

However, computing the Hessian can become quite a difficult task. Moreover, what is needed is the *inverse* of the Hessian. In neural networks we have to repeat this computation on each new iteration. Consequently, many techniques have been proposed to approximate the second-order information contained in the Hessian using certain heuristics.

Pseudo-Newton methods are variants of Newton's method that work with a simplified form of the Hessian matrix [48]. The non-diagonal elements are all set to zero and only the diagonal elements are computed, that is, the second derivatives of the form $\partial^2 E(\mathbf{w})/\partial w_i^2$. In that case equation (8.8) simplifies (for each component of the weight vector) to

$$w_i^{(k+1)} = w_i^{(k)} - \frac{\nabla_i E(\mathbf{w})}{\partial^2 E(\mathbf{w})/\partial w_i^2}. \tag{8.9}$$

No matrix inversion is necessary and the computational effort involved in finding the required second partial derivatives is limited. In Sect. 8.4.3 we show how to perform this computation efficiently.

Pseudo-Newton methods work well when the error function has a nice quadratic form, otherwise care should be exercised with the corrections, since a small second-order partial derivative can lead to extremely large corrections.

8.4.1 Quickprop

In this section we consider an algorithm which tries to take second-order information into account but follows a rather simple approach: only one-dimensional minimization steps are taken and information about the curvature of the error function in the update directions is obtained from the current and the last partial derivative of the error function in this direction.

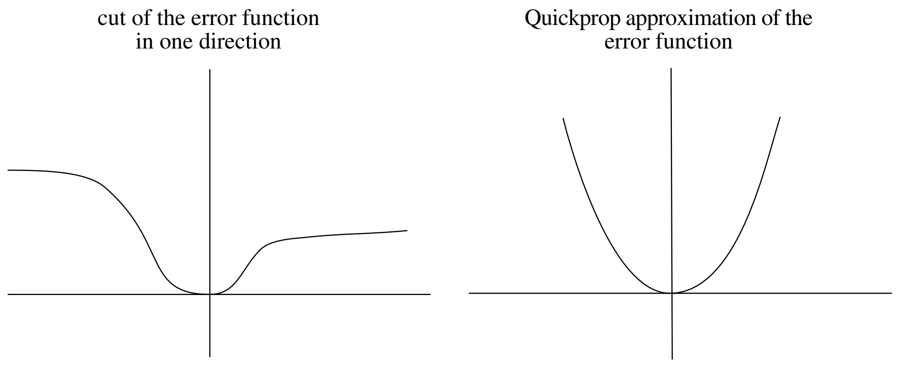

cut of the error function in one direction
Quickprop approximation of the error function

Fig. 8.18. Local approximation of Quickprop

Quickprop is based on independent optimization steps for each weight. A quadratic one-dimensional approximation of the error function is used. The update term for each weight at the k-th step is given by

$$\Delta^{(k)} w_i = \Delta^{(k-1)} w_i \left(\frac{\nabla_i E^{(k)}}{\nabla_i E^{(k-1)} - \nabla_i E^{(k)}} \right), \tag{8.10}$$

where it is assumed that the error function has been computed at steps $(k-1)$ and k using the weight difference $\Delta^{(k-1)} w_i$, obtained from a previous Quickprop or an standard gradient descent step.

Note that if we rewrite (8.10) as

$$\Delta^{(k)} w_i = -\frac{\nabla_i E^{(k-1)}}{(\nabla_i E^{(k)} - \nabla_i E^{(k)})/\Delta^{(k-1)} w_i} \tag{8.11}$$

then the weight update in (8.11) is of the same form as the weight update in (8.9). The denominator is just a discrete approximation to the second-order derivative $\partial^2 E(\mathbf{w})/\partial w_i^2$. Quickprop is therefore a discrete pseudo-Newton method that uses so-called *secant steps.*

According to the value of the derivatives, Quickprop updates may become very large. This is avoided by limiting $\Delta^{(k)} w_i$ to a constant times $\Delta^{(k-1)}$. See [130] for more details on the algorithm and the handling of different problematic situations. Since the assumptions on which Quickprop is based are more far-fetched than the assumptions used by, for example, Rprop, it is not surprising that Quickprop has some convergence problems with certain tasks and requires careful handling of the weight updates [341].

8.4.2 QRprop

Pfister and Rojas proposed an algorithm that adaptively switches between the Manhattan method used by Rprop and local one-dimensional secant steps like those used by Quickprop [340, 341]. Since the resulting algorithm is a hybrid of Rprop and Quickprop it was called QRprop.

QRprop uses the individual learning rate strategy of Rprop if two consecutive error function gradient components $\nabla_i E^{(k)}$ and $\nabla_i E^{(k-1)}$ have the same sign or one of these components equals zero. This produces a fast approach to a region of minimum error. If the sign of the gradient changes, we know that we have overshot a local minimum in this specific weight direction, so now a second-order step (a Quickprop step) is taken. If we assume that in this direction the error function is independent from all the other weights, a step based on a quadratic approximation will be far more accurate than just stepping half way back as it is (indirectly) done by Rprop. Since the error function depends on all weights and since the quadratic approximation will be better the closer the two investigated points lie together, QRprop constrains the size of the secant step to avoid large oscillations of the weights. In summary:

i) As long as $\nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} > 0$ holds, Rprop steps are performed because we assume that a local minimum lies ahead.

ii) If $\nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} < 0$, which suggests that a local minimum has been overshot, then, unlike Rprop, neither the individual learning rate γ_i nor the weight w_i are changed. A "marker" is defined by setting $\nabla_i E^{(k)} := 0$ and the secant step is performed in the subsequent iteration.

iii) If $\nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} = 0$, this means that either a marker was set in the previous step, or one of the gradient components is zero because a local minimum has been directly hit. In both cases we are near a local minimum and the algorithm performs a second-order step. The secant approximation is done using the gradient information provided by $\nabla_i E^{(k)}$ and $\nabla_i E^{(k-2)}$. The second-order approximation is used even when $\nabla_i E^{(k)} \cdot \nabla_i E^{(k-2)} > 0$. Since we know that we are near a local minimum (and very likely we have already overshot it in the previous step), the second-order approximation is still a better choice than just stepping halfway back.

iv) In the secant step the quadratic approximation

$$q_i := |\nabla_i E^{(k)})/(\nabla_i E^{(k)} - \nabla_i E^{(k-2)})|$$

is constrained to a certain interval to avoid very large or very small updates.

Therefore, the k-th iteration of the algorithm consists of the following steps:

Step 1: Update the individual learning rates

```
if $(\nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} = 0)$ then
    if $(\nabla_i E^{(k)} \neq (\nabla_i E^{(k-2)})$ then
        $q_i = \max\left(d, \min\left(1/u, \left|\frac{\nabla_i E^{(k)}}{\nabla_i E^{(k)} - \nabla_i E^{(k-2)}}\right|\right)\right)$
    else
        $q_i = 1/u$
    endif
endif
```

$$\gamma_i^{(k)} = \begin{cases} \min(u \cdot \gamma_i^{(k-1)},\ \gamma_{max}) & \nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} > 0 \\ \gamma_i^{(k-1)} & \text{if } \ \nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} < 0 \\ \max(q_i \cdot \gamma_i^{(k-1)},\ \gamma_{min}) & \nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} = 0 \end{cases}$$

Step 2: Update the weights

$$w_i^{(k+1)} = \begin{cases} w_i^{(k)} - \gamma_i^{(k)} \cdot \operatorname{sgn}(\nabla_i E^{(k)}) & \text{if } \nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} \geq 0 \\ w_i^{(k)} & \text{otherwise} \end{cases}$$

If $(\nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} < 0)$ set $\nabla_i E^{(k)} := 0$.

The constants d, u, γ_{min}, and γ_{max} must be chosen in advance as for other adaptive steps methods. QRprop has shown to be an efficient algorithm that can outperform both of its original algorithmic components. Table 8.2 shows the speedup obtained with QRprop relative to Rprop for the Carnegie Mellon benchmarks [341].

Table 8.2. Speedup of QRprop relative to Rprop

benchmark	speedup
sonar signals	1.01
vowels	1.33
vowels (decorrelated)	1.30
NETtalk (200 words)	1.02
protein structure	1.14
digits	1.29
average	1.18

8.4.3 Second-order backpropagation

In this section we introduce *second-order backpropagation*, a method to efficiently compute the Hessian of a linear network of one-dimensional functions. This technique can be used to get explicit symbolic expressions or numerical approximations of the Hessian and could be used in parallel computers to improve second-order learning algorithms for neural networks. Methods for the determination of the Hessian matrix in the case of multilayered networks have been studied recently [58].

We show how to efficiently compute the elements of the Hessian matrix using a graphical approach, which reduces the whole problem to a computation by inspection. Our method is more general than the one developed in [58] because arbitrary topologies can be handled. The only restriction we impose on the network is that it should contain no cycles, i.e., it should be of the feed-forward type. The method is of interest when we do not want to derive analytically the Hessian matrix each time the network topology changes.

Second-order derivatives

We investigate the case of second-order derivatives, that is, expressions of the form $\partial^2 F/\partial w_i \partial w_j$, where F is the network function as before and w_i and w_j are network's weights. We can think of each weight as a small potentiometer and we want to find out what happens to the network function when the resistance of both potentiometers is varied.

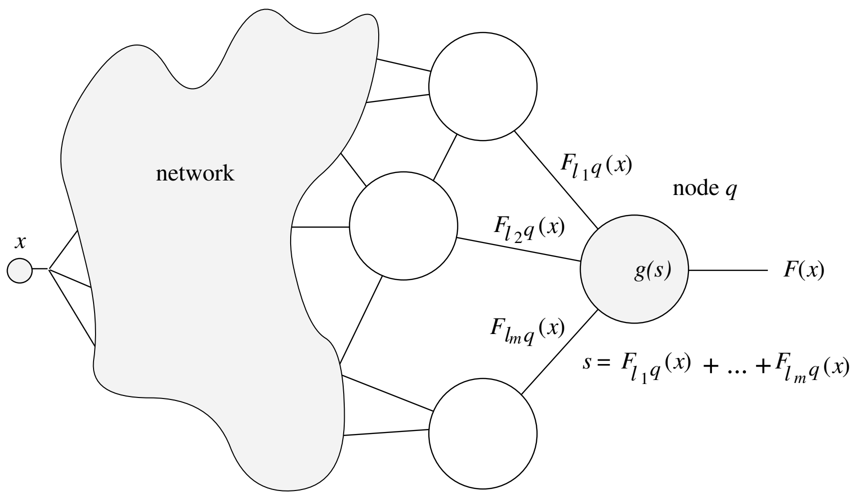


Fig. 8.19. Second-order computation

Figure 8.19 shows the general case. Let us assume, without loss of generality, that the input to the network is the one-dimensional value x. The network function F is computed at the output node with label q (shown shaded) for the given input value. We can also think of the inputs to the output node as network functions computed by subnetworks of the original network. Let us call these functions $F_{\ell_1 q}, F_{\ell_2 q}, \ldots, F_{\ell_m q}$. If the one-dimensional function at the output node is g, the network function is the composition

$$F(x) = g(F_{\ell_1 q}(x) + F_{\ell_2 q}(x) + \ldots + F_{\ell_m q}(x)).$$

We are interested in computing $\partial^2 F(x)/\partial w_i \partial w_j$ for two given network weights w_i and w_j. Simple differential calculus tells us that

$$\begin{aligned}\frac{\partial^2 F(x)}{\partial w_i \partial w_j} &= g''(s)\frac{\partial s}{\partial w_i}\frac{\partial s}{\partial w_j} \\ &\qquad + g'(s)\left(\frac{\partial^2 F_{\ell_1 q}(x)}{\partial w_i \partial w_j} + \cdots + \frac{\partial^2 F_{\ell_m q}(x)}{\partial w_i \partial w_j}\right),\end{aligned}$$

where $s = F_{\ell_1 q}(x) + F_{\ell_2 q}(x) + \ldots + F_{\ell_m q}(x)$. This means that the desired second-order partial derivative consists of two terms: the first is the second derivative of g evaluated at its input multiplied by the partial derivatives of the sum of the m subnetwork functions $F_{\ell_1 q}, \ldots, F_{\ell_m q}$, once with respect to w_i and once with respect to w_j. The second term is the first derivative of g multiplied by the sum of the second-order partial derivatives of each subnetwork function with respect to w_i and w_j. We call this term the *second-order correction.* The recursive structure of the problem is immediately obvious. We already have an algorithm to compute the first partial derivatives of any network function with respect to a weight. We just need to use the above expression in a recursive manner to obtain the second-order derivatives we want.

We thus extend the feed-forward labeling phase of the backpropagation algorithm in the following manner: at each node which computes a one-dimensional function f we will store *three* values: $f(x), f'(x)$ and $f''(x)$, where x represents the input to this node. When looking for the second-order derivatives we apply the recursive strategy given above. Figure 8.20 shows the main idea:

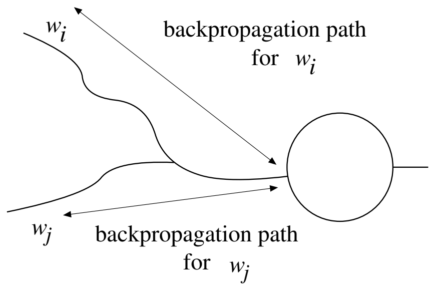

w_i
backpropagation path for w_i
w_j
backpropagation path for w_j

Fig. 8.20. Intersecting paths to a node

- Perform the feed-forward labeling step in the usual manner, but store additionally at each node the second derivative of the node's function evaluated at its input
- Select two weights w_i and w_j and an output node whose associated network function we want to derive. The second-order partial derivative of the network function with respect to these weights is the product of the stored g'' value with the backpropagation path value from the output node up to weight w_i and with the backpropagation path value from the output node up to weight w_j. If the backpropagation paths for w_i and w_j intersect, a second-order correction is needed which is equal to the stored value of g' multiplied by the sum of the second-order derivative with respect to w_i and w_j of all subnetwork function inputs to the node which belong to intersecting paths.

This looks like an intricate rule, but it is again the chain rule for second-order derivatives expressed in a recursive manner. Consider the multilayer perceptron shown in Figure 8.21. A weight w_{ih} in the first layer of weights and a weight w_{jm} in the second layer can only interact at the output node m. The second derivative of F_m with respect to w_{ih} and w_{jm} is just the stored value f'' multiplied by the stored output of the hidden unit j and the backpropagation path up to w_{ih}, that is, $w_{hm}h'x_i$. Since the backpropagation paths for w_{ih} and w_{jm} do not intersect, this is the required expression. This is also the expression found analytically by Bishop [1993].

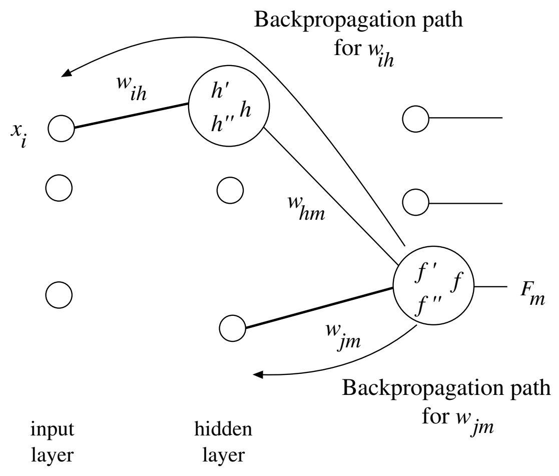


Fig. 8.21. Multilayer perceptron

In the case where one weight lies in the backpropagation path of another, a simple adjustment has to be made. Let us assume that weight w_{ik} lies in the backpropagation path of weight w_j. The second-order backpropagation algorithm is performed as usual and the backward computation proceeds up to the point where weight w_{ik} transports an input to a node k for which a second-order correction is needed. Figure 8.22 shows the situation. The information transported through the edge with weight w_{ik} is the subnetwork function F_{ik}. The second-order correction for the node with primitive function g is

$$g' \frac{\partial^2 F_{ik}}{\partial w_{ik} \partial wj} = g' \frac{\partial^2 w_{ik} F_i}{\partial w_{ik} \partial wj},$$

but this is simply

$$g' \frac{\partial F_i}{\partial w_j},$$

since the subnetwork function F_i does not depend on w_{ik}. Thus, the second-order backpropagation method must be complemented by the following rule:

- If the second-order correction to a node k with activation function g involves a weight w_{ik} (that is, a weight directly affecting node k) and a node w_j, the second-order correction is just g' multiplied by the backpropagation path value of the subnetwork function F_i with respect to w_j.

Explicit calculation of the Hessian

For the benefit of the reader, we put together all the pieces of what we call second-order backpropagation in this section. We consider the case of a single input pattern into the network, since the more general case is easy to handle.

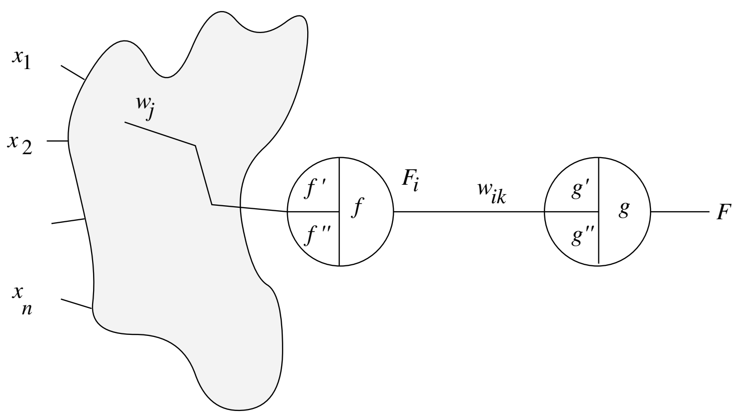


Fig. 8.22. The special case

Algorithm 8.4.1 *Second-order backpropagation*

i) Extend the neural network by adding nodes which compute the squared difference of each component of the output and the expected target values. Collect all these differences at a single node whose output is the error function of the network. The activation function of this node is the identity.

ii) Label all nodes in the feed-forward phase with the result of computing $f(x)$, $f'(x)$, and $f''(x)$, where x represents the global input to each node and f its associated activation function.

iii) Starting from the error function node in the extended network, compute the second-order derivative of E with respect to two weights w_i and w_j, by proceeding recursively in the following way:

  iii.1) The second-order derivative of the output of a node G with activation function g with respect to two weights w_i and w_j is the product of the stored g'' value with the backpropagation path values between w_i and the node G and between w_j and the node G. A second-order correction is needed if both backpropagation paths intersect.

  iii.2) The second-order correction is equal to the product of the stored g' value with the sum of the second-order derivative (with respect to w_i and w_j) of each node whose output goes directly to G and which belongs to the intersection of the backpropagation paths of w_i and w_j.

  iii.3) In the special case that one of the weights, for example, w_i, connects node h directly to node G, the second-order correction is just g' multiplied by the backpropagation path value of the subnetwork function F_h with respect to w_j.

Example of second-order backpropagation

Consider the network shown in Figure 8.23, commonly used to compute the XOR function. The left node is labeled 1, the right node 2. The input values x and y are kept fixed and we are interested in the second-order partial derivative of the network function $F_2(x, y)$ with respect to the weights w_1 and w_2.

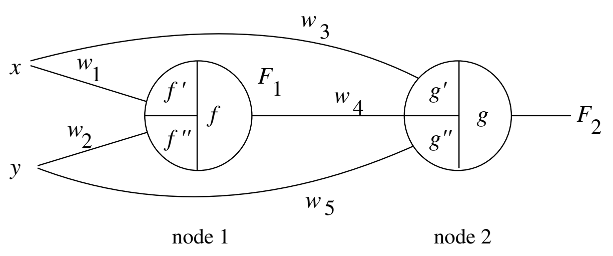


Fig. 8.23. A two unit network

By mere inspection and using the recursive method mentioned above, we see that the first term of $\partial^2 F_2/\partial w_1 \partial w_2$ is the expression

$$g''(w_3x + w_5y + w_4f(w_1x + w_2y))(w_4f'(w_1x + w_2y)x)(w_4f'(w_1x + w_2y)y).$$

In this expression $(w_4f'(w_1x+w_2y)x)$ is the backpropagation path value from the output of the node which computes the function f, including multiplication by the weight w_4 (that is the subnetwork function w_4F_1), up to the weight w_1. The term $(w_4f'(w_1x+w_2y)y)$ is the result of backpropagation for w_4F_1 up to w_2. The second-order correction needed for computation of $\partial^2 F_2/\partial w_1 \partial w_2$ is

$$g'(w_3x + w_5y + w_4f(w_1x + w_2y))\frac{\partial^2 w_4F_1}{\partial w_1 \partial w_2}.$$

Since it is obvious that

$$\frac{\partial^2 w_4F_1}{\partial w_1 \partial w_2} = w_4\frac{\partial^2 F_1}{\partial w_1 \partial w_2} = w_4f''(w_1x + w_2y)xy$$

we finally get

$$\begin{aligned}\frac{\partial^2 F_2}{\partial w_1 \partial w_2} = {} & g''(w_3x + w_5y + w_4f(w_1x + w_2y)) \\ & \times (w_4f'(w_1x + w_2y)x)(w_4f'(w_1x + w_2y)y) \\ & + g'(w_3x + w_5y + w_4f(w_1x + w_2y))w_4f''(w_1x + w_2y)xy.\end{aligned}$$

The reader can *visually* check the following expression:

$$\frac{\partial^2 F_2}{\partial w_1 \partial w_5} = g''(w_3x + w_5y + w_4f(w_1x + w_2y))(w_4f'(w_1x + w_2y)x)y.$$

In this case no second-order correction is needed, since the backpropagation paths up to w_1 and w_5 do not intersect.

A final example is the calculation of the whole Hessian matrix for the network shown above (Figure 8.23). We omit the error function expansion and compute the Hessian of the network function F_2 with respect to the network's five weights. The labelings of the nodes are f, f', and f'' computed over the input $w_1x + w_2y$, and g, g', g'' computed over the input $w_4 f(w_1x + w_2y) + w_3x + w_5y$. Under these assumptions the components of the upper triangular part of the Hessian are the following:

$$\begin{aligned}
H_{11} &= g''w_4^2 {f'}^2 x^2 + g'w_4 f''x^2 \\
H_{22} &= g''w_4^2 {f'}^2 y^2 + g'w_4 f''y^2 \\
H_{33} &= g''x^2 \\
H_{44} &= g''f^2 \\
H_{55} &= g''y^2 \\
H_{12} &= g''w_4^2 {f'}^2 xy + g'w_4 f''xy \\
H_{13} &= g''w_4 f'x^2 \\
H_{14} &= g''w_4 f'xf + g'f'x \\
H_{15} &= g''w_4 f'xy \\
H_{23} &= g''w_4 f'yx \\
H_{24} &= g''w_4 f'yf + g'f'y \\
H_{25} &= g''w_4 f'y^2 \\
H_{34} &= g''xf \\
H_{35} &= g''xy \\
H_{45} &= g''yf
\end{aligned}$$

All these results were obtained by simple inspection of the network shown in Figure 8.23. Note that the method is totally general in the sense that each node can compute a different activation function.

Some conclusions

With some experience it is easy to compute the Hessian matrix even for convoluted feed-forward topologies. This can be done either symbolically or numerically. The importance of this result is that once the recursive strategy has been defined it is easy to implement in a computer. It is the same kind of difference as the one existing between the chain rule and the backpropagation algorithm. The first one gives us the same result as the second, but backpropagation tries to organize the data in such a way that redundant computations are avoided. This can also be done for the method described here. Calculation of the Hessian matrix involves repeated computation of the same terms. In this case the network itself provides us with a data structure in which we can store partial results and with which we can organize the computation. This explains why standard and second-order backpropagation are also of interest for computer algebra systems. It is not very difficult to program the method described here in a way that minimizes the number of arithmetic operations needed. The key observation is that the backpropagation path values can be stored to be used repetitively and that the nodes in which the backpropagation paths of different weights intersect need to be calculated only once. It is then possible to optimize computation of the Hessian using graph traversing algorithms.

A final observation is that computing the diagonal of the Hessian matrix involves only local communication in a neural network. Since the backpropagation path to a weight intersects itself in its whole length, computation of the second partial derivative of the associated network function of an output unit with respect to a given weight can be organized as a recursive backward computation over this path. Pseudo-Newton methods [48] can profit from this computational locality.

8.5 Relaxation methods

The class of relaxation methods includes all those techniques in which the network weights are perturbed and the new network error is compared directly to the previous one. Depending on their relative magnitudes a decision is taken regarding the subsequent iteration steps.

8.5.1 Weight and node perturbation

Weight perturbation is a learning strategy which has been proposed to avoid calculating the gradient of the error function at each step. A discrete approximation to the gradient is made at each iteration by taking the initial weight vector $\mathbf{w}$ in weight space and the value $E(w)$ of the error function for this combination of parameters. A small perturbation β is added to the weight w_i. The error $E(\mathbf{w}')$ at the new point $\mathbf{w}'$ in weight space is computed and the weight w_i is updated using the increment

$$\varDelta w_i = -\gamma \frac{E(w') - E(w)}{\beta}.$$

This step is repeated iteratively, randomly selecting the weight to be updated. The discrete approximation to the gradient is especially important for VLSI chips in which the learning algorithm is implemented with minimal hardware additional to that needed for the feed-forward phase [216].

Another alternative which can provide faster convergence is perturbing not a weight, but the output o_i of the i-th node by $\varDelta o_i$. The difference $E - E'$ in the error function is computed and if it is positive, the new error E' could be achieved with the output $o_i + \Delta o_i$ for the i-th node. If the activation function is a sigmoid, the desired weighted input to node i is $\sum_{k=1}^{m} w'_k x_k = s^{-1}(o_i + \Delta o_i)$. If the previous weighted input was $\sum_{k=1}^{m} w_k x_k$, then the new weights are given by

$$w'_k = w_k \frac{s^{-1}(o_i + \Delta o_i)}{\sum_{k=1}^{m} w_k x_k} \qquad \text{for } k = 1, \ldots, m.$$

The weights are updated in proportion to their relative size. To avoid always keeping the same proportions, a stochastic factor can be introduced or a node perturbation step can be alternated with a weight perturbation step.

8.5.2 Symmetric and asymmetric relaxation

According to the analysis we made in Sect. 7.3.3 of two layered networks trained with backpropagation, the backpropagated error up to the output layer can be written as

$$\delta^{(2)} = \mathbf{D}_2 \mathbf{e},$$

where $\mathbf{e}$ is the column vector whose components are the derivatives of the corresponding components of the quadratic error, and $\mathbf{D}_2$ is a diagonal matrix as defined in Chap. 7. We can try to reduce the magnitude of the error $\mathbf{e}$ to zero by adjusting the matrix $\mathbf{W}_2$ in a single step. Since the desired target vector is $\mathbf{t}$, the necessary weighted input at the output nodes is $s^{-1}(\mathbf{t})$. This means that we want the equation

$$\mathbf{o}^{(1)} \mathbf{W}_2 = s^{-1}(\mathbf{t})$$

to hold for all the p possible input patterns. If we arrange all vectors $\mathbf{o}^{(1)}$ as the rows of a $p \times k$ matrix $\mathbf{O}_1$ and all targets as the rows of a $p \times m$ matrix $\mathbf{T}$, we are looking for the matrix $\mathbf{W}_2$ for which

$$\mathbf{O}_1 \mathbf{W}_2 = s^{-1}(\mathbf{T}) \tag{8.12}$$

holds. In general this matrix equation may have no solution for $\mathbf{W}_2$, but we can compute the matrix which minimizes the quadratic error for this equality. We show in Sects. 9.2.4 and **??** that if $\mathbf{O}_1^+$ is the so-called *pseudoinverse* of $\mathbf{O}_1$, then $\mathbf{W}_2$ is given by

$$\mathbf{W}_2 = \mathbf{O}_1^+ s^{-1}(\mathbf{T}).$$

After computing $\mathbf{W}_2$ we can ask what is the matrix $\mathbf{O}_1$ which minimizes the quadratic deviation from equality in (8.12). We compute intermediate targets for the hidden units, which are given by the rows of the matrix

$$\mathbf{O}'_1 = s^{(-1)}(\mathbf{T}) \mathbf{W}_2^+.$$

The new intermediate targets can now be used to obtain an update for the matrix $\mathbf{W}_1$ of weights between input sites and hidden units. The pseudoinverse can be computed with the method discussed in Sect. 9.2.4.

Note that this method is computationally intensive for every "pseudoinverse" step. The weight corrections are certainly much more accurate than in other algorithms but these high-powered iterations demand many computations for each of the two matrices. If the error function can be approximated nicely with a quadratic function, the algorithm converges very fast. If the input data is highly redundant, then the pseudoinverse step can be very inefficient when compared to on-line backpropagation, for example.

The algorithm described is an example of *symmetric* relaxation, since both the targets **T** and the outputs of the hidden units are determined in a back and forth kind of approach.

8.5.3 A final thought on taxonomy

The contents of this chapter can be summarized using Figure 8.24. The fast variations of backpropagation have been divided into two columns: gradient descent and relaxation methods. Algorithms in the first column use information about the error function's partial derivatives. Algorithms in the second column try to adjust the weights to fit the problem in a stochastic way or solving a linear subproblem.

The three rows in Figure 8.24 show the kinds of derivative used. First-order algorithms work with the first partial derivatives, second-order algorithms with the second partial derivatives. In between we have adaptive first-order methods that from first-order information extract an approximation to parts of the Hessian matrix.

Standard backpropagation is a first-order gradient descent method. However, since on-line backpropagation does not exactly follow the gradient's direction, it also partially qualifies as a relaxation method. The adaptive step methods (DBD, Rprop, etc.) are also a combination of gradient descent and relaxation. Quickprop is an algorithm which approximates second-order information but which updates the weights separately using a kind of relaxation approach.

The adaptive first-order methods can be also divided in two groups: in the first a single global learning rate is used, in the second there is a learning rate for each weight. The conjugate gradient methods of numerical analysis rely also on a first-order approximation of second-order features of the error function.

8.6 Historical and bibliographical remarks

This survey of fast learning algorithms is by no means complete. We have disregarded constructive algorithms of the type reviewed in Chap. 14. We have only given a quick overview of some of the main ideas that have been developed in the last decade. There is a large amount of literature on nonlinear optimization that should be reviewed by anyone wishing to improve current

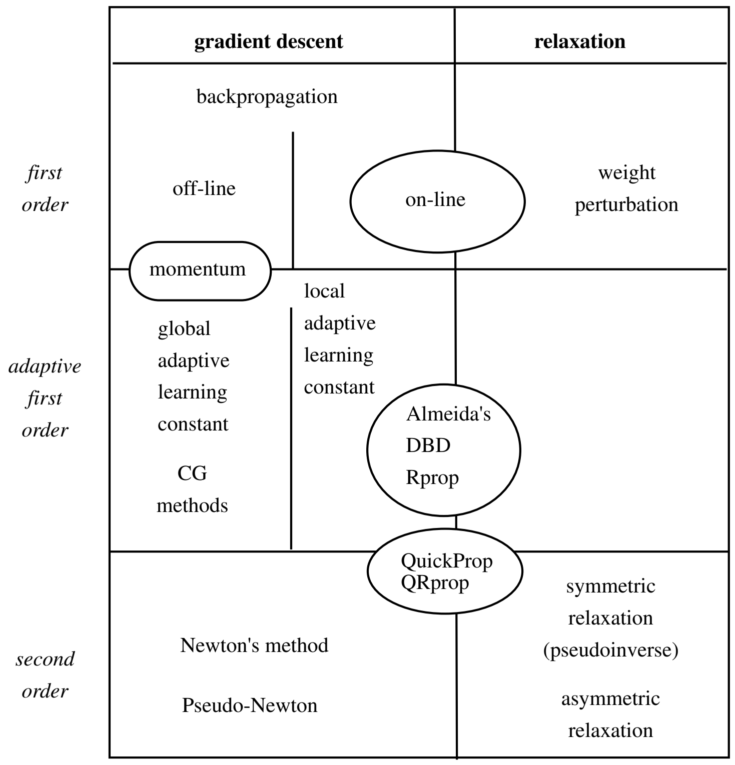


Fig. 8.24. Taxonomy of learning algorithms

learning methods for multilayer networks. Classical CG algorithms and some variants developed specially for neural networks are interesting in this respect [315].

A different approach has been followed by Karayiannis and Venetsanopoulos who use a type of what is called a continuation method [235]. The network is trained to minimize a given measure of error, which is iteratively changed during the computation. One can start solving a linear regression problem and introduce the nonlinearities slowly. The method could possibly be combined with some of the other fast learning methods.

A factor which has traditionally hampered direct comparisons of learning algorithms is the wide variety of benchmarks used. Only in a few cases have large learning problems taken from public domain databases been used. Some efforts to build a more comprehensive set of benchmarks have been announced.

Exercises

1. Prove that the value of the learning constant given by equation (8.5) in fact produces exact learning of the j input pattern to a linear associator. Derive the expression for ℓ^2.
2. Show how to construct the linear transformation that maps an ellipsoidal data distribution to a sphere. Assume that the training set consists of N points (Sect. 8.2.4).
3. Compare Rprop and Quickprop using a small benchmark (for example the 8-bit parity problem).
4. Train a neural network using: a) Backpropagation and a piecewise linear approximation of the sigmoid; b) A table of values of the sigmoid and its derivative.

9

Statistics and Neural Networks

9.1 Linear and nonlinear regression

Feed-forward networks are used to find the best functional fit for a set of input-output examples. Changes to the network weights allow fine-tuning of the network function in order to detect the optimal configuration. However, two complementary motivations determine our perception of what optimal means in this context. On the one hand we expect the network to map the known inputs as exactly as possible to the known outputs. But on the other hand the network must be capable of *generalizing*, that is, unknown inputs are to be compared to the known ones and the output produced is a kind of interpolation of learned values. However, good generalization and minimal reproduction error of the learned input-output pairs can become contradictory objectives.

9.1.1 The problem of good generalization

Figure 9.1 shows the problem from another perspective. The dots in the graphic represent the training set. We are looking for a function capable of mapping the known inputs into the known outputs. If linear approximation is used, as in the figure, the error is not excessive and new unknown values of the input x are mapped to the regression line.

Figure 9.2 shows another kind of functional approximation using linear splines which can reproduce the training set without error. However, when the training set consists of experimental points, normally there is some noise in the data. Reproducing the training set exactly is not the best strategy, because the noise will also be reproduced. A linear approximation as in Figure 9.1 could be a better alternative than the exact fit of the training data shown in Figure 9.2. This simple example illustrates the two contradictory objectives of functional approximation: minimization of the training error but also minimization of the error of yet unknown inputs. Whether or not the training set can be

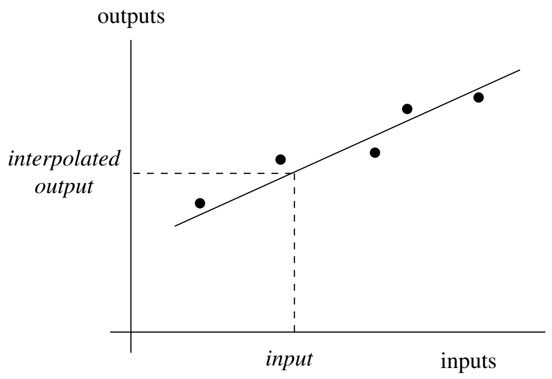


Fig. 9.1. Linear approximation of the training set

learned exactly depends on the number of degrees of freedom available to the network (number of weights) and the structure of the manifold from which the empirical data is extracted. The number of degrees of freedom determines the *plasticity* of the system, that is, its capability of approximating the training set. Increasing the plasticity helps to reduce the training error but can increase the error on the test set. Decreasing the plasticity excessively can lead to a large training and test error.

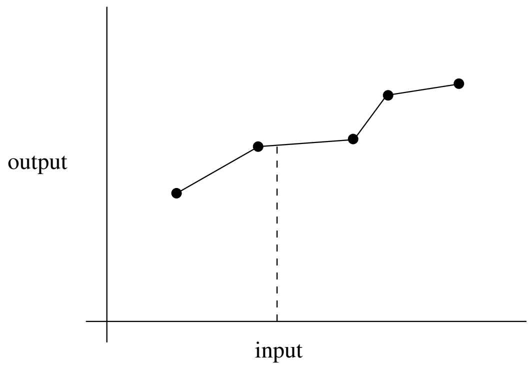


Fig. 9.2. Approximation of the training set with linear splines

There is no universal method to determine the optimal number of parameters for a network. It all depends on the structure of the problem at hand. The best results can be obtained when the network topology is selected taking into account the known interrelations between input and output (see Chap. 14). In the example above, if a theoretical analysis leads us to conjecture a linear correspondence between input and output, the linear approximation would be the best although the polylinear approximation has a smaller training error.

This kind of functional approximation to a given training set has been studied by statisticians working in the field of linear and nonlinear regression. The backpropagation algorithm is in some sense only a numerical method for statistical approximation. Analysis of the linear case can improve our understanding of this connection.

9.1.2 Linear regression

Linear associators were introduced in Chap. 5: they are computing units which just add their weighted inputs. We can also think of them as the integration part of nonlinear units. For the n-dimensional input $(x_1, x_2, \ldots, x_n)$ the output of a linear associator with weight vector $(w_1, w_2, \ldots, w_n)$ is $y = w_1x_1 + \cdots + w_nx_n$. The output function represents a hyperplane in $(n+1)$-dimensional space. Figure 9.3 shows the output function of a linear associator with two inputs. The learning problem for such a linear associator is to reproduce the output of the input vectors in the training set. The points corresponding to the training set are shown in black in Figure 9.3. The parameters of the hyperplane must be selected to minimize the error, that is, the distance from the training set to the hyperplane. The backpropagation algorithm can be used to find them.

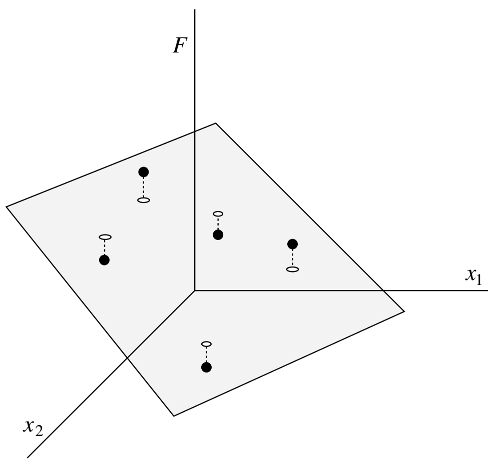


Fig. 9.3. Learning problem for a linear associator

Consider a training set $T = \{(\mathbf{x}^1, a_1), \ldots, (\mathbf{x}^m, a_m)\}$ for a linear associator, where the inputs $\mathbf{x}^1, \ldots, \mathbf{x}^m$ are n-dimensional vectors and the outputs $a_1, \ldots, a_m$ real numbers. We are looking for the weight vector $(w_1, \ldots, w_n)$ which minimizes the quadratic error

$$E = \frac{1}{2}\left[\left(a_1 - \sum_{i=1}^{n} w_i x_i^1\right)^2 + \cdots + \left(a_m - \sum_{i=1}^{n} w_i x_i^m\right)^2\right] \tag{9.1}$$

where x_i^j denotes the i-th component of the j-th input vector. The components of the gradient of the error function are

$$\frac{\partial E}{\partial w_j} = -\left(a_1 - \sum_{i=1}^{n} w_i x_i^1\right) x_j^1 - \cdots - \left(a_m - \sum_{i=1}^{n} w_i x_i^m\right) x_j^m \tag{9.2}$$

for $j = 1, 2, \ldots, n$. The minimum of the error function can be found analytically by setting $\nabla E = \mathbf{0}$ or iteratively using gradient descent. Since the error function is purely quadratic the global minimum can be found starting from randomly selected weights and making the correction $\Delta w_j = -\gamma \partial E/\partial w_j$ at each step.

Figure 9.4 shows the B-diagram for a linear associator. The training vector $\mathbf{x}^1$ has been used to compute the error E_1. The partial derivatives $\partial E/\partial w_1, \ldots, \partial E/\partial w_n$ can be computed using a backpropagation step.

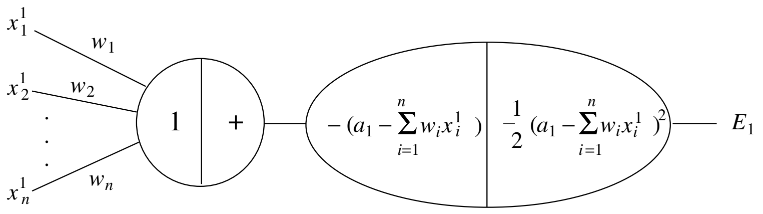

x_1^1
w_1
x_2^1
w_2
x_n^1
w_n
1
+
$-(a_1 - \sum_{i=1}^{n} w_i x_i^1)$
$\frac{1}{2}(a_1 - \sum_{i=1}^{n} w_i x_i^1)^2$
E_1

Fig. 9.4. Backpropagation network for the linear associator

The problem of finding optimal weights for a linear associator and for a given training set T is known in statistics as *multiple linear regression*. We are looking for constants $w_0, w_1, \ldots, w_n$ such that the y values can be computed from the x values:

$$y_i = w_0 + w_1 x_1^i + w_2 x_2^i + \cdots + w_n x_n^i + \varepsilon_i\,,$$

where ε_i represents the approximation error (note that we now include the constant w_0 in the approximation). The constants selected should minimize the total quadratic error $\sum_{i=1}^{n} \varepsilon_i^2$. This problem can be solved using algebraic methods. Let X denote the following $m \times (n+1)$ matrix:

$$\mathbf{X} = \begin{pmatrix} 1 & x_1^1 & \cdots & x_n^1 \\ 1 & x_1^2 & \cdots & x_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^m & \cdots & x_n^m \end{pmatrix}$$

The rows of the matrix consist of the extended input vectors. Let **a**, **w** and ε denote the following vectors:

$$\mathbf{a} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix} \qquad \mathbf{w} = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{pmatrix} \qquad \boldsymbol{\varepsilon} = \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_m \end{pmatrix}$$

The vector **w** must satisfy the equation $\mathbf{a} = \mathbf{X}\mathbf{w} + \boldsymbol{\varepsilon}$, where the norm of the vector $\boldsymbol{\varepsilon}$ must be minimized. Since

$$\|\boldsymbol{\varepsilon}\|^2 = (\mathbf{a} - \mathbf{X}\mathbf{w})^{\mathrm{T}}(\mathbf{a} - \mathbf{X}\mathbf{w})$$

the minimum of the norm can be found by equating the derivative of this expression with respect to **w** to zero:

$$\frac{\partial}{\partial \mathbf{w}}(\mathbf{a} - \mathbf{X}\mathbf{w})^{\mathrm{T}}(\mathbf{a} - \mathbf{X}\mathbf{w}) = -2\mathbf{X}^{\mathrm{T}}\mathbf{a} + 2\mathbf{X}^{\mathrm{T}}\mathbf{X}\mathbf{w} = \mathbf{0}.$$

It follows that $\mathbf{X}^{\mathrm{T}}\mathbf{X}\mathbf{w} = \mathbf{X}^{\mathrm{T}}\mathbf{a}$ and if the matrix $\mathbf{X}^{\mathrm{T}}\mathbf{X}$ is invertible, the solution to the problem is given by

$$\mathbf{w} = \left(\mathbf{X}^{\mathrm{T}}\mathbf{X}\right)^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{a}.$$

9.1.3 Nonlinear units

Introducing the sigmoid as the activation function changes the form of the functional approximation produced by a network. In Chap. 7 we saw that the form of the functions computed by the sigmoidal units corresponds to a smooth step function. As an example in Figure 9.5 we show the continuous output of two small networks of sigmoidal units. The first graphic corresponds to the network in Figure 6.2 which can compute an approximation to the XOR function when sigmoidal units are used. The output of the network is approximately 1 for the inputs $(1, 0)$ and $(0, 1)$ and approximately 0 for the inputs $(0, 0)$ and $(1, 1)$. The second graph corresponds to the computation of the NAND function with three sigmoidal units distributed in two layers.

Much more complicated functions can be produced with networks which are not too elaborate. Figure 9.8 shows the functions produced by a network with three and four hidden units and a single output unit. Small variations of the network parameters can produce widely differing shapes and this leads us to suspect that any continuous function could be approximated in this manner, if only enough hidden units are available. The number of foldings of the functions corresponds to the number of hidden units. In this case we have a situation similar to when polynomials are used to approximate experimental data—the degree of the polynomial determines the number of degrees of freedom of the functional approximation.

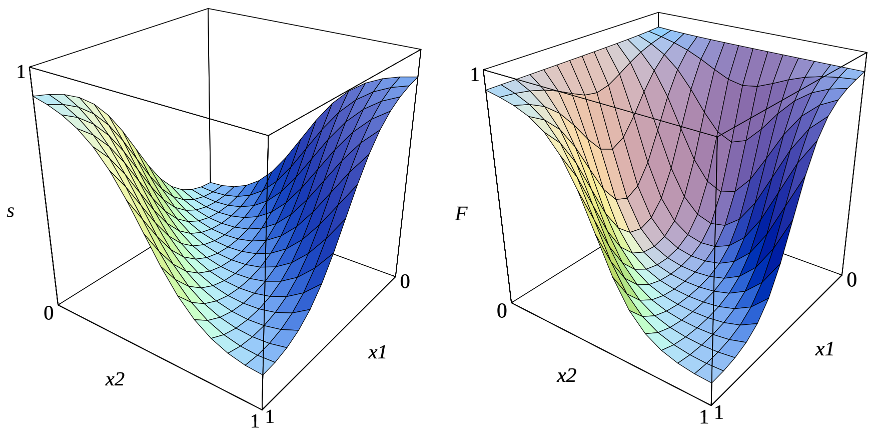


Fig. 9.5. Output of networks for the computation of XOR (left) and NAND (right)

Logistic regression

Backpropagation applied to a linear association problem finds the parameters of the optimal linear regression. If a sigmoid is computed at the output of the linear associator, we are dealing with the conventional units of feed-forward networks.

There is a type of nonlinear regression which has been applied in biology and economics for many years called *logistic regression*. Let the training set T be $\{(\mathbf{x}^1, a_1), (\mathbf{x}^2, a_2), \ldots, (\mathbf{x}^m, a_m)\}$, where the vectors $\mathbf{x}^i$ are n-dimensional. A sigmoidal unit is to be trained with this set. We are looking for the n-dimensional weight vector $\mathbf{w}$ which minimizes the quadratic error

$$E = \sum_{i=1}^{m} (a_i - s(\mathbf{w} \cdot \mathbf{x}^i))^2 \,,$$

where s denotes the sigmoid function. Backpropagation solves the problem directly by minimizing E. An approximation can be found using the tools of linear regression by inverting the sigmoid and minimizing the new error function

$$E' = \sum_{i=1}^{m} (s^{-1}(a_i) - \mathbf{w} \cdot \mathbf{x}^i)^2.$$

Since the a_i are constants this step can be done at the beginning so that a linear associator has to approximate the outputs

$$a'_i = s^{-1}(a_i) = \ln\left(\frac{a_i}{1 - a_i}\right), \quad \text{for } i = 1, \ldots, m. \tag{9.3}$$

All the standard machinery of linear regression can be used to solve the problem. Equation (9.3) is called the *logit transformation* [34]. It simplifies the approximation problem but at a cost. The logit transformation modifies the weight given to the individual deviations. If the target value is 0.999 and the sigmoid output is 0.990, the approximation error is 0.009. If the logit transformation is used, the approximation error for the same combination is 2.3 and can play a larger role in the computation of the optimal fit. Consequently, backpropagation is a type of nonlinear regression [323] which solves the approximation problem in the original domain and is therefore more precise.

9.1.4 Computing the prediction error

The main issue concerning the kind of functional approximation which can be computed with neural networks is to obtain an estimate of the prediction error when new values are presented to the network. The case of linear regression has been studied intensively and there are closed-form formulas for the expected error and its variance. In the case of nonlinear regression of the kind which neural networks implement, it is very difficult, if not impossible, to produce such analytic formulas. This difficulty also arises when certain kinds of statistics are extracted from empirical data. It has therefore been a much-studied problem. In this subsection we show how to apply some of these statistical methods to the computation of the expected generalization error of a network.

One might be inclined to think that the expected generalization error of a network is just the square root of the mean squared training error. If the training set consists of N data points and E is the total quadratic error of the network over the training set, the generalization error $\tilde{E}$ could be set to

$$\tilde{E} = \sqrt{E/N}.$$

This computation, however, would tend to underestimate the true generalization error because the parameters of the network have been adjusted to deal with exactly this data set and could be biased in favor of its elements. If many additional input-output pairs that do not belong to the training set are available, the generalization error can be computed directly. New input vectors are fed into the network and the mean quadratic deviation is averaged over many trials. Normally, this is not the case and we want to use all of the available data to train the network *and* to predict the generalization error.

The *bootstrap* method, proposed by Efron in 1979, deals with exactly this type of statistical problem [127]. The key observation is that existent data can be used to adjust a predictor (such as a regression line), yet it also tells us something about the distribution of the future expected inputs. In the real world we would perform linear regression and compute the generalization error using new data not included in the training set. In the *bootstrap world* we try to imitate this situation by sampling randomly from the existing data to create different training sets.

Here is how the bootstrap method works: assume that a data set $X = \{x_1, x_2, \ldots, x_n\}$ is given and that we compute a certain statistic $\hat{\theta}$ with this data. This number is an estimate of the true value θ of the statistic over the whole population. We would like to know how reliable is $\hat{\theta}$ by computing its standard deviation. The data is produced by an unknown probability distribution F. The bootstrap assumption is that we can approximate this distribution by randomly sampling from the set X with replacement. We generate a new data set X^* in this way and compute the new value of the statistics which we call $\hat{\theta}^*$. This procedure can be repeated many times with many randomly generated data sets. The standard deviation of $\hat{\theta}$ is approximated by the standard deviation of $\hat{\theta}^*$.

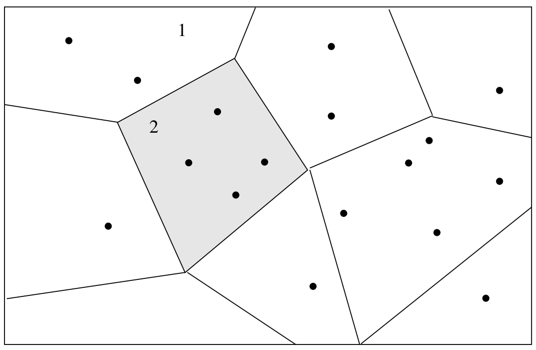


Fig. 9.6. Distribution of data in input space

Figure 9.6 graphically shows the idea behind the bootstrap method. The experimental data comes from a certain input space. If we want to compute some function over the whole input space (for example if we want to find the centroid of the complete input domain), we cannot because we only have some data points, but we can produce an estimate assuming that the distribution of the data is a good enough approximation to the actual probability distribution. The figure shows several regions where the data density is different. We approximate this varying data density by sampling *with replacement* from the known data. Region 2 in the figure will then be represented twice as often as region 1 in the generated samples. Thus our computations use not the unknown probability distribution F, but an approximation $\hat{F}$. This is the "plug-in principle": the empirical distribution $\hat{F}$ is an estimate of the true distribution F. If the approximation is good enough we can derive more information from the data, such as the standard deviation of function values computed over the empirical data set.

Algorithm 9.1.1 *Bootstrap algorithm*

i) Select N independent bootstrap samples $\mathbf{x}^{*1}, \mathbf{x}^{*2}, \ldots, \mathbf{x}^{*N}$ each consisting of n data values selected with replacement from X.

ii) Evaluate the desired statistic S corresponding to each bootstrap sample,

$$\hat{\theta}^*(b) = S(\mathbf{x}^{*b}) \qquad b = 1, 2, \ldots, N.$$

iii) Estimate the standard error $\hat{s}_N$ by the sample standard deviation of the N replications

$$\hat{s}_N = \left(\sum_{b=1}^{N} [\hat{\theta}^*(b) - \tilde{\theta}\,]^2 / (N-1) \right)^{1/2}$$

where $\tilde{\theta} = \sum_{b=1}^{N} \hat{\theta}^*(b)/N$.

In the case of functional approximation the bootstrap method can be applied in two different ways, but the simpler approach is the following. Assume that a neural network has been trained to approximate the function φ associated with the training set $T = \{(\mathbf{x}_1, \mathbf{t}_1), \ldots, (\mathbf{x}_m, \mathbf{t}_m)\}$ of m input-output pairs. We can compute a better estimate of the expected mean error by generating N different bootstrap training sets. Each bootstrap training set is generated by selecting m input-output pairs from the original training set randomly and with replacement. The neural network is trained always using the same algorithm and stop criterion. For each network trained we compute:

- The mean squared error Q_i^* for the i-th bootstrap training set,
- The mean squared error for the original data, which we call Q_i^0.

The standard deviation of the Q_i^* values is an approximation to the true standard deviation of our function fit.

In general, Q_i^* will be lower than Q_i^0, because the training algorithm adjusts the parameters optimally for the training set at hand. The *optimism* in the computation of the expected error is defined as

$$O = \frac{1}{B} \sum_{i=1}^{B} (Q_i^0 - Q_i^*).$$

The idea of this definition is that the original data set is a fair representative of the whole input space and the unknown sample distribution F, whereas the bootstrap data set is a fair representative of a generic training set extracted from input space. The optimism O gives a measure of the degree of underestimation present in the mean squared error originally computed for a training set.

There is a complication in this method which does not normally arise when the statistic of interest is a generic function. Normally, neural networks training is nondeterministic because the error function contains several global minima which can be reached when gradient descent learning is used. Retraining of networks with different data sets could lead to several completely different solutions in terms of the weights involved. This in turn can lead to disparate estimates of the mean quadratic deviation for each bootstrap data set. However, if we want to analyze what will happen in general when the given network is trained with data coming from the given input space, this is precisely the right thing to do because we never know at which local minima training stopped. If we want to analyze just one local minimum we must ensure that training always converges to *similar* local minima of the error function (only similar because the shape of the error function depends on the training set used and different training sets have different local minima). One way to do this was proposed by Moody and Utans, who trained a neural network using the original data set and then used the weights found as initial weights for the training of the bootstrap data sets [319]. We expect gradient descent to converge to nearby solutions for each of the bootstrap data sets. Especially important is that with the bootstrap method we can compute confidence intervals for the neural approximation [127].

9.1.5 The jackknife and cross-validation

A relatively old statistical technique which can be considered a predecessor of the bootstrap method is the *jackknife*. As in the bootstrap, new data samples are generated from the original data, but in a much simpler manner. If n data points are given, one is left out, the statistic of interest is computed with the remaining $n-1$ points and the end result is the average over the n different data sets. Figure 9.7 shows a simple example comparing the bootstrap with the jackknife for the case of three data points, where the desired statistic is the centroid position of the data set. In the case of the bootstrap there are 10 possible bootstrap sets which lead to 10 different computed centroids (shown in the figure as circles with their respective probabilities). For the jackknife there are 3 different data sets (shown as ellipses) and centroids. The average of the bootstrap and jackknife "populations" coincide in this simple example. The d-jackknife is a refinement of the standard method: instead of leaving one point out of the data set, d different points are left out and the statistic of interest is computed with the remaining data points. Mean values and standard deviations are then computed as in the bootstrap.

In the case of neural networks *cross-validation* has been in use for many years. For a given training set T some of the input-output pairs are reserved and are not used to train the neural network (typically 5% or 10% of the data). The trained network is tested with these reserved input-output pairs and the observed average error is taken as an approximation of the true mean squared error over the input space. This estimated error is a good approximation if both training and test set fully reflect the probability distribution of the data in input space. To improve the results *k-fold cross-validation* can be used. The data set is divided into k random subsets of the same size. The network is trained k times, each time leaving one of the k subsets out of the training set and testing the mean error with the subset which was left out. The average of

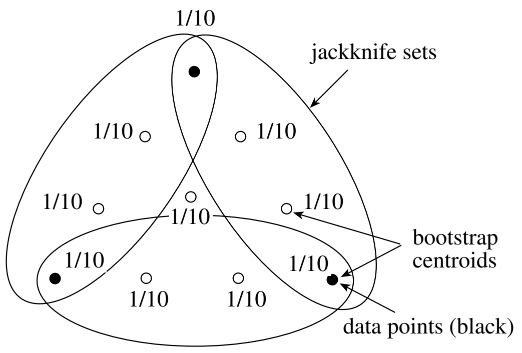


Fig. 9.7. Comparison of the bootstrap and jackknife sampling points for $n = 3$

the k computed mean quadratic errors is our estimate of the expected mean quadratic error over the whole of input space. As in the case of the bootstrap, the initial values of the weights for each of the k training sets can be taken from previous results using the complete data set, a technique called *nonlinear cross-validation* by Moody and Utans [319], or each network can be trained with random initial weights. The latter technique will lead to an estimation of the mean quadratic deviation over different possible solutions of the given task.

The bootstrap, jackknife, and cross-validation are all methods in which raw computer power allows us to compute confidence intervals for statistics of interest. When applied to neural networks, these methods are even more computationally intensive because training the network repetitively consumes an inordinate amount of time. Even so, if adequate parallel hardware is available the bootstrap or cross-validation provides us with an assessment of the reliability of the network results.

9.1.6 Committees of networks

The methods for the determination of the mean quadratic error discussed in the previous section rely on training several networks with the same basic structure. If so much computing power is available, the approximation capabilities of an ensemble of networks is much better than just using one of the trained networks. The combination of the outputs of a group of neural networks has received several different names in the literature, but the most suggestive denomination is undoubtedly *committees* [339].

Assume that a training set of m input-output pairs $(\mathbf{x}^1, t_1), \ldots, (\mathbf{x}^m, t_m)$ is given and that N networks are trained using this data. For simplicity we consider n-dimensional input vectors and a single output unit. Denote by f_i the network function computed by the i-th network, for $i = 1, \ldots, N$. The network function f produced by the committee of networks is defined as

$$f = \frac{1}{N} \sum_{i=1}^{N} f_i.$$

The rationale for this averaging over the network functions is that if each one of the approximations is biased with respect to some part of input space, an average over the ensemble of networks can reduce the prediction error significantly. For each network function f_i we can compute an m-dimensional vector $\mathbf{e}^i$ whose components are the approximation error of the function f_i for each input-output pair. The quadratic approximation error Q of the ensemble function f is

$$Q = \sum_{i=1}^{m} \left(t_i - \frac{1}{N} \sum_{j=1}^{N} f_j(\mathbf{x}^i) \right)^2 .$$

This can be written in matrix form by defining a matrix $\mathbf{E}$ whose N rows are the m components of each error vector $\mathbf{e}^i$:

$$\mathbf{E} = \begin{pmatrix} e_1^1 & e_2^1 & \cdots & e_m^1 \\ \vdots & \vdots & \ddots & \vdots \\ e_1^N & e_2^N & \cdots & e_m^N \end{pmatrix}$$

The quadratic error of the ensemble is then

$$Q = \left| \frac{1}{N}(1,1,\ldots,1)\mathbf{E} \right|^2 = \frac{1}{N^2}(1,1,\ldots,1)\mathbf{E}\mathbf{E}^{\mathrm{T}}(1,1,\ldots,1)^{\mathrm{T}} \qquad (9.4)$$

The matrix $\mathbf{EE}^{\mathrm{T}}$ is the correlation matrix of the error residuals. If each function approximation produces uncorrelated error vectors, the matrix $\mathbf{EE}^{\mathrm{T}}$ is diagonal and the i-th diagonal element Q_i is the sum of quadratic deviations for each functional approximation, i.e., $Q_i = \|\mathbf{e}^i\|^2$. In this case

$$Q = \frac{1}{N} \left(\frac{1}{N}(Q_1 + \cdots + Q_N) \right),$$

and this means that the total quadratic error of the ensemble is smaller by a factor $1/N$ than the average of the quadratic errors of the computed functional approximations. Of course this impressive result holds only if the assumption of uncorrelated error residuals is true. This happens mostly when N is not too large. In some cases even $N = 2$ or $N = 3$ can lead to significant improvement of the approximation capabilities of the combined network [339].

If the quadratic errors are not uncorrelated, that is if $\mathbf{EE}^{\mathrm{T}}$ is not symmetric, a weighted combination of the N functions f_i can be used. Denote the i-th weight by w_i. The ensemble functional approximation f is now

$$f = \sum_{i=1}^{N} w_i f_i.$$

The weights w_i must be computed in such a way as to minimize the expected quadratic deviation of the function f for the given training set. With the same definitions as before and with the constraint $w_1 + \cdots + w_N = 1$ it is easy to see that equation (9.4) transforms to

$$Q = \frac{1}{N^2}(w_1, w_2, \ldots, w_N)\mathbf{E}\mathbf{E}^{\mathrm{T}}(w_1, w_2, \ldots, w_N)^{\mathrm{T}}.$$

The minimum of this expression can be found by differentiating with respect to the weight vector $(w_1, w_2, \ldots, w_N)$ and setting the result to zero. But because of the constraint $w_1 + \cdots + w_N = 1$ a Lagrange multiplier λ has to be included so that the function to be minimized is

$$\begin{aligned} Q' &= \frac{1}{N^2}\mathbf{w}\mathbf{E}\mathbf{E}^{\mathrm{T}}\mathbf{w}^{\mathrm{T}} + \lambda(1,1,\ldots,1)\mathbf{w}^{\mathrm{T}} \\ &= \frac{1}{N^2}\mathbf{w}\mathbf{E}\mathbf{E}^{\mathrm{T}}\mathbf{w}^{\mathrm{T}} + \lambda\mathbf{1}\mathbf{w}^{\mathrm{T}} \end{aligned}$$

where $\mathbf{1}$ is a row vector with all its N components equal to 1. The partial derivative of Q' with respect to $\mathbf{w}$ is set to zero and this leads to

$$\frac{1}{N^2}\mathbf{w}\mathbf{E}\mathbf{E}^{\mathrm{T}} + \lambda\mathbf{1} = 0\,.$$

If the matrix $\mathbf{E}\mathbf{E}^{\mathrm{T}}$ is invertible this leads to

$$\mathbf{w} = -\lambda N^2\mathbf{1}(\mathbf{E}\mathbf{E}^{\mathrm{T}})^{-1}.$$

From the constraint $\mathbf{w}\mathbf{1}^{\mathrm{T}} = 1$ we deduce

$$\mathbf{w}\mathbf{1}^{\mathrm{T}} = -\lambda N^2\mathbf{1}(\mathbf{E}\mathbf{E}^{\mathrm{T}})^{-1}\mathbf{1}^{\mathrm{T}} = 1\,,$$

and therefore

$$\lambda = -\frac{1}{N^2\mathbf{1}(\mathbf{E}\mathbf{E}^{\mathrm{T}})^{-1}\mathbf{1}^{\mathrm{T}}}.$$

The final optimal set of weights is

$$\mathbf{w} = \frac{\mathbf{1}(\mathbf{E}\mathbf{E}^{\mathrm{T}})^{-1}}{\mathbf{1}(\mathbf{E}\mathbf{E}^{\mathrm{T}})^{-1}\mathbf{1}^{\mathrm{T}}},$$

assuming that the denominator does not vanish. Notice that the constraint $\mathbf{w}\mathbf{1}^{\mathrm{T}}$ is introduced only to simplify the analysis of the quadratic error.

This method can become prohibitive if the matrix EE^{T} is ill-conditioned or if its computation requires too many operations. In that case an adaptive method can be used. Note that the vector of weights can be learned using a Lagrange network of the type discussed in Chap. 7.

9.2 Multiple regression

Backpropagation networks are a powerful tool for function approximation. Figure 9.8 shows the graphs of the output function produced by four networks. The graph on the lower right was produced using four hidden units, the other using three. It can be seen that such a small variation in the topology of the network leads to an increase in the plasticity of the network function. The number of degrees of freedom of a backpropagation network, and therefore its plasticity, depends on the number of weights, which in turn are a function of the number of hidden units. How many of them should be used to solve a given problem? The answer is problem-dependent: in the ideal case, the network function should not have more degrees of freedom than the data itself, because otherwise there is a danger of overtraining the network.

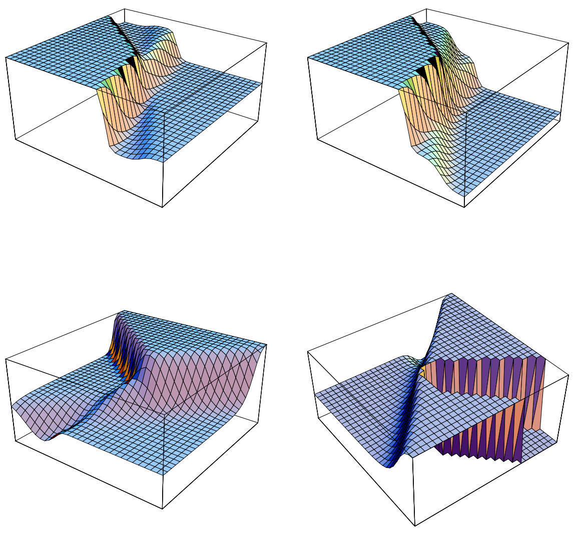


Fig. 9.8. Network functions of networks with one hidden layer

In this section we look at the role of the hidden layer, considering the interplay between layers in a network, but first of all we develop a useful visualization of the multiple regression problem.

9.2.1 Visualization of the solution regions

Consider a training set $T = \{(\mathbf{x}^1, a_1), (\mathbf{x}^2, a_2), \ldots, (\mathbf{x}^m, a_m)\}$ consisting of n-dimensional inputs and scalar outputs. We are looking for the best approximate solution to the system of equations

$$s(\mathbf{x}^i \cdot \mathbf{w}) = a_i, \quad \text{for } i = 1, 2, \ldots, m, \tag{9.5}$$

where s denotes, as usual, the sigmoid and $\mathbf{w}$ is the weight vector for a linear associator. If the outputs a_i are real and lie in the interval $(0, 1)$, equation (9.5) can be rewritten as

$$\mathbf{x}^i \cdot \mathbf{w} = s^{-1}(a_i), \quad \text{for } i = 1, 2, \ldots, m. \tag{9.6}$$

The m equations in (9.6) define m hyperplanes in weight space. If all hyperplanes meet at a common point $\mathbf{w}_0$, then this weight vector is the exact solution of the regression problem and the approximation error is zero. If there is no common intersection, there is no exact solution to the problem. When $m > n$, that is, when the number of training pairs is higher than the dimension of weight space, the hyperplanes may not meet at a single common point. In this case we must settle for an approximate solution of the regression problem.

Consider now the polytopes defined in weight space by the system of equations (9.6). First consider the case of a linear associator (eliminating the sigmoid and its inverse). The training equations are in this case

$$\mathbf{x}^i \cdot \mathbf{w} = a_i, \quad \text{for} \quad i = 1, 2, \ldots, m. \tag{9.7}$$

If there is no common intersection of the m hyperplanes, we look for the weight vector $\mathbf{w}'$ which minimizes the quadratic norms ε_i^2, where

$$\varepsilon_i = \mathbf{x}^i \cdot \mathbf{w}' - a_i, \quad \text{for} \quad i = 1, 2, \ldots, m. \tag{9.8}$$

A two-dimensional example can serve to illustrate the problem. Consider the three lines ℓ_1, ℓ_2 and ℓ_3 shown in Figure 9.9. The three do not intersect at a common point but the point with the minimum total distance to the three lines is α. This is also the site in weight space which can be found by using linear regression.

Important for the solution of the regression problem is that the square of the distance of α to each line is a quadratic function. The sum of quadratic functions is also quadratic and its minimization presents no special numerical problem. The point α in Figure 9.9 lies at a global minimum (in this case unique) of the error function.

The systems of equations (9.6) and (9.7) are very similar, but when the sigmoid is introduced the approximation error is given by

$$E = \sum_{i=1}^{m} \left(s(\mathbf{x}^i \cdot \mathbf{w}) - a_i \right)^2 , \tag{9.9}$$

which is not a quadratic function of $\mathbf{w}$. Suboptimal local minima can now appear.

Figure 9.10 shows the form of the error function for the same example of Figure 9.9 when the sigmoid is introduced. The error function now has three different local minima and the magnitude of the error can be different in any of them. Gradient descent would find one of the three minima, but not necessarily the best.

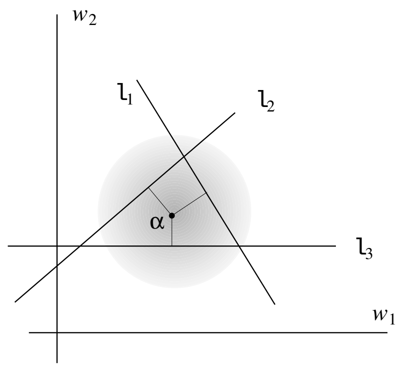


Fig. 9.9. Point of minimal distance to three lines

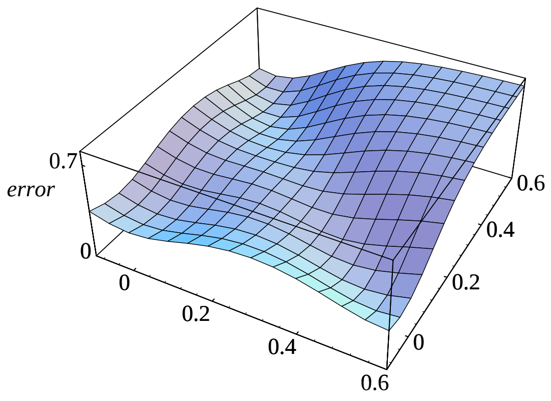


Fig. 9.10. Local minima of the error function

9.2.2 Linear equations and the pseudoinverse

Up to this point we have only considered the regression problem for individual linear associators. Consider now a network with two layers of weights, as shown in Figure 9.11. Assume that the training set consists of the n-dimensional input vectors $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^m$ and the k-dimensional output vectors $\mathbf{y}^1, \mathbf{y}^2, \ldots, \mathbf{y}^m$. Let $\mathbf{W}_1$ be the weight matrix between the input sites and the hidden layer (with the same conventions as in Chap. 7, but without bias terms). Let $\mathbf{W}_2$ be the weight matrix between hidden and output layer. If all units in the network are linear associators, the output for the input $\mathbf{x}$ is $\mathbf{x}\mathbf{W}_1\mathbf{W}2$. The weight matrix $\mathbf{W} = \mathbf{W}_1\mathbf{W}_2$ could be used in a network without a hidden layer and the output would be the same. The hidden layer plays a role *only* if the hidden units introduce some kind of nonlinearity in the computation.

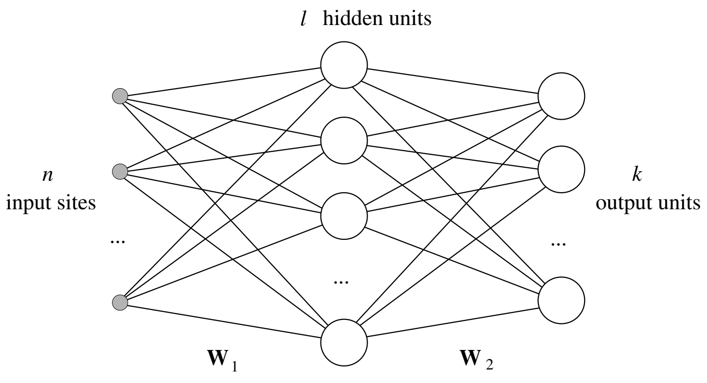


Fig. 9.11. Multilayer network

Assume that the hidden layer consists of ℓ units. Let $\mathbf{Y}$ denote the $m \times k$ matrix whose rows are the row vectors $\mathbf{y}^i$, for $i = 1, \ldots, m$. Let $\mathbf{Z}$ denote the $m \times \ell$ matrix whose rows are each one of the vectors produced by the hidden layer for the inputs $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^m$. The output of the network can be written as

$$\mathbf{Y} = \mathbf{Z}\mathbf{W}_2 \,.$$

It is interesting to point out that this condition is always fulfilled if the m rows of the matrix $\mathbf{Z}$ are linearly independent. In that case there exists a matrix called the *pseudoinverse* $\mathbf{Z}^+$ such that $\mathbf{Z}\mathbf{Z}^+ = \mathbf{I}$, where $\mathbf{I}$ denotes the $m \times m$ identity matrix (we will discuss the properties of the pseudoinverse in Chap. **??**). Setting $\mathbf{W}_2 = \mathbf{Z}^+\mathbf{Y}$ we get $\mathbf{Y} = \mathbf{Z}\mathbf{W}_2$ because $\mathbf{Z}\mathbf{Z}^+\mathbf{Y} = \mathbf{Y}$ [349]. If the input vectors can be mapped to linearly independent vectors in the hidden layer the learning problem has a solution. This requires that $m \leq \ell$. This loose upper bound on the number of hidden units is not better than when each hidden unit acts as a feature detector for each input vector. Genuine learning problems can usually be solved with smaller networks.

9.2.3 The hidden layer

We can give the nonlinearity in the hidden layer a geometric interpretation. The computation between input sites and hidden layer corresponds to a linear transformation followed by a nonlinear "compression", that is, the evaluation of a squashing function at the hidden units. Let us first consider units with a step function as nonlinearity, that is, perceptrons. Assume that a unit in the hidden layer has the associated weight vector $\mathbf{w}_1$. All vectors in input space close enough to vector $\mathbf{w}_1$ are mapped to the same vector in feature space, for example the vector (0,1) in a network with two hidden units. A cone around vector $\mathbf{w}_1$ acts as its basin of attraction. Figure 9.12 also shows the basin of attraction of vector $\mathbf{w}_2$. Vectors close to $\mathbf{w}_2$ are mapped to the vector $(1, 0)$ in feature space.

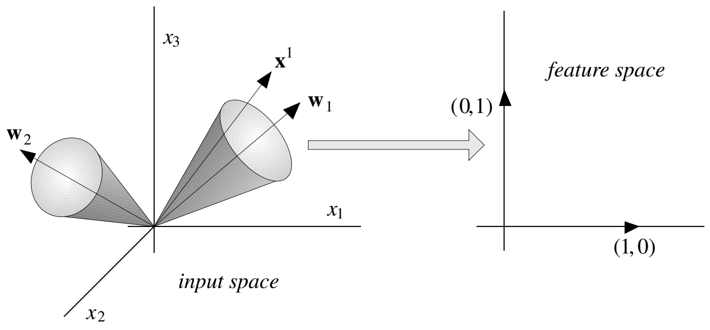


Fig. 9.12. Mapping input space into feature space

If all computed vectors in feature space are linearly independent, it is possible to find a matrix $\mathbf{W}_2$ that produces any desired output. If not a step function but a sigmoid is used as nonlinearity, the form of the basins of attraction changes and the vectors in feature space can be a combination of the basis vectors.

We can summarize the functioning of a network with a hidden layer in the following way: the hidden layer defines basins of attraction in input space so that the input vectors are mapped to vectors in feature space. Then, it is necessary to solve a linear regression problem between hidden and output layer in order to minimize the quadratic error over the training set.

9.2.4 Computation of the pseudoinverse

If m input vectors are mapped to m ℓ-dimensional linearly independent vectors $\mathbf{z}^1, \mathbf{z}^2, \ldots, \mathbf{z}^m$ in feature space, the backpropagation algorithm can be used to find the $\ell \times m$ matrix $\mathbf{Z}^+$ for which $\mathbf{Z}\mathbf{Z}^+ = \mathbf{I}$ holds. In the special case $m = \ell$ we are looking for the inverse of the square matrix $\mathbf{Z}$. This can be done using gradient descent. We will come back to this problem in Chap. **??**. Here we only show how the linear regression problem can be solved.

The network in Figure 9.13 can be used to compute the inverse of $\mathbf{Z}$. The training input consists of the m vectors $\mathbf{z}^1, \mathbf{z}^2, \ldots, \mathbf{z}^m$ and the training output of the m rows of the $m \times m$ identity matrix (the elements of the identity matrix are represented by Kronecker's delta, and the j-th component of the network output for the i-th training vector by o_j^i). When the input vectors are linearly independent the error function has a unique global minimum, which can be found using the backpropagation algorithm. The weight matrix $\mathbf{W}$ converges to $\mathbf{Z}^{-1}$.

If the m input vectors are not linearly independent, backpropagation nevertheless finds a minimum of the error function. This corresponds to the problem in which we do not look for the common intersection of hyperplanes but for the point with the minimal cumulative distance to all of them (Figure 9.9). If the minimum is unique the network finds the pseudoinverse $\mathbf{Z}^+$ of the matrix

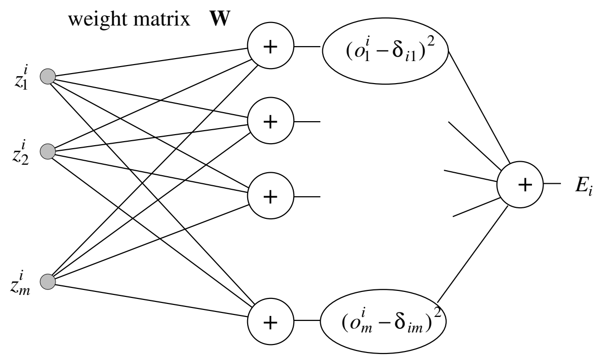


Fig. 9.13. Network for the computation of the inverse

Z. If the minimum is not unique (this can happen when $m < \ell$) it is necessary to minimize the norm of the weight matrix in order to find the pseudoinverse [14]. This can be done by adding a *decay term* to the backpropagation weight updates. The modified updates are given by

$$\Delta w_{ij} = -\gamma \frac{\partial E}{\partial w_{ij}} - \kappa w_{ij},$$

where κ and γ denote constants. The decay term tends to lower the magnitude of each weight—it corresponds to the negative partial derivative of w_{ij}^2 with respect to w_{ij}.

9.3 Classification networks

Multilayered neural networks have become a popular tool for a growing spectrum of applications. They are being applied in robotics, in speech or pattern recognition tasks, in coding problems, etc. It has been said that certain problems are theory-driven whereas others are data-driven. In the first class of problems theory predominates, in the latter there is much data but less theoretical understanding. Neural networks can discover statistical regularities and keep adjusting parameters even in a changing environment. It is interesting to look more closely at some applications where the statistical properties of neural networks become especially valuable.

There are many applications in which a certain input has to be classified as belonging to one of k different classes. The input is a certain measurement which we want to label in a predetermined way. This kind of problem can be solved by a *classification network* with k output units. Given a certain input vector $\mathbf{x}$ we expect the network to set the output line associated with the correct classification of $\mathbf{x}$ to 1 and the others to 0. In this section we discuss how to train such networks and then we look more closely at the range of output values being produced.

9.3.1 An application: NETtalk

Speech synthesis systems have been commercially available for quite a number of years now. They transform a string of characters into a string of phonemes by applying some linguistic transformation rules. The number of such rules is rather large and their interaction is not trivial [196].

In the 1980s Sejnowski and Rosenberg developed a backpropagation network that was able to synthesize speech of good quality without applying explicit linguistic transformations [396]. The authors used a backpropagation network composed of seven groups of 29 input sites, 80 hidden and 26 output units. The text to be pronounced by the system is scanned using a sliding window of seven characters. Each one of the characters is coded as one of 29 possible letters (one input line is set to 1 the other 28 to 0). Consequently there are $7 \times 29 = 203$ input sites. The network must produce the correct phoneme for the pronunciation of the character in the middle of the window, taking into account the three characters of context to the left and to the right. The network was connected to an electronic speech synthesizer capable of synthesizing 26 phonemes (later variants of NETtalk have used more phonemes). The network contains around 18,000 weights which must be found by the learning algorithm. We expect the network to be able to extract the statistical regularities from the training set by itself.

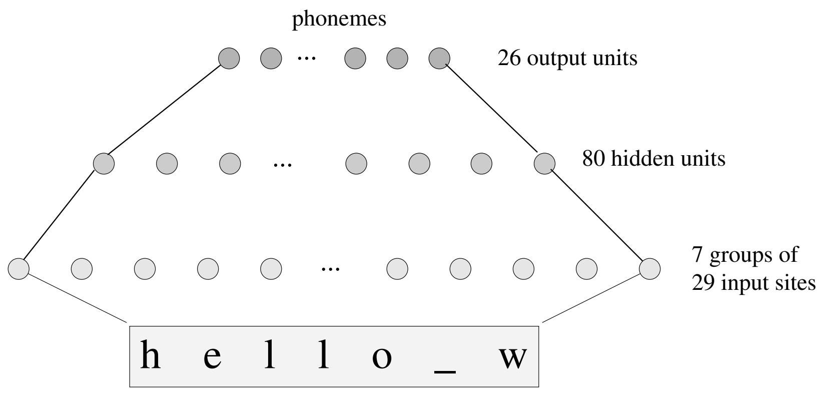


Fig. 9.14. The NETtalk architecture

The training set consists of a corpus of several hundred words, together with their phonetic transcription. The network is trained to produce a 1 at the output unit corresponding to the right phoneme. After training, an unknown text is scanned and the output units are monitored. At each time step only the unit with the maximum output level is selected. Surprisingly, the speech generated is of comparable quality to that produced by much more intricate rule-based systems.

Sejnowski and Rosenberg also looked at the proficiency of the network at different learning stages. At the beginning of learning the network made some of the same errors as children do when they learn to speak. Damaging some of the weights produced some specific deficiencies. Analyzing the code produced by the hidden units, the authors determined that some of them had implicitly learned some of the known linguistic rules.

NETtalk does not produce exactly ones or zeros, and the pronounced phoneme is determined by computing the maximum of all output values. It is interesting to ask what the produced output values stand for. One possibility is that it indeed represents the probability that each phoneme could be the correct one. However, the network is trained with binary values only, so that the question to be answered is: how do classifier networks learn probabilities?

9.3.2 The Bayes property of classifier networks

It is now well known that neural networks trained to classify an n-dimensional input x in one out of M classes can actually learn to compute the *a posteriori* probabilities that the input x belongs to each class. Several proofs of this fact, differing only in the details, have been published [65, 365], but they can be simplified. In this section we offer a shorter proof of the probability property of classifier neural networks proposed by Rojas [377].

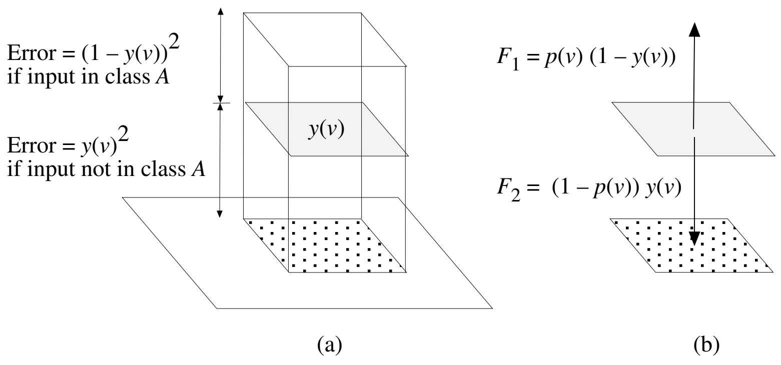


Fig. 9.15. The output $y(v)$ in a differential volume

Part (a) of Figure 9.15 shows the main idea of the proof. Points in an input space are classified as belonging to a class A or its complement. This is the first simplification: we do not have to deal with more than one class. In classifier networks, there is one output line for each class C_i, $i = 1, \ldots, M$. Each output C_i is trained to produce a 1 when the input belongs to class i, and otherwise a 0. Since the expected total error is the sum of the expected individual errors of each output, we can minimize the expected individual errors independently. This means that we need to consider only one output line and whether it should produce a 1 or a 0.

Proposition 13. *A classifier neural network perfectly trained and with enough plasticity can learn the a posteriori probability of an empirical data set.*

Proof. Assume that input space is divided into a lattice of differential volumes of size dv, each one centered at the n-dimensional point v. If at the output representing class A the network computes the value $y(v) \in [0,1]$ for any point x in the differential volume $V(v)$ centered at v, and denoting by $p(v)$ the probability $p(A|x \in V(v))$, then the total expected quadratic error is

$$E_A = \sum_V \{p(v)(1-y(v))^2 + (1-p(v))y(v)^2\}dv,$$

where the sum runs over all differential volumes in the lattice. Assume that the values $y(v)$ can be computed independently for each differential volume. This means that we can independently minimize each of the terms of the sum. This is done by differentiating each term with respect to the output $y(v)$ and equating the result to zero:

$$-2p(v)(1-y(v)) + 2(1-p(v))y(v) = 0.$$

From this expression we deduce $p(v) = y(v)$, that is, the output $y(v)$ which minimizes the error in the differential region centered at v is the a posteriori probability $p(v)$. In this case the expected error is

$$p(v)(1-p(v))^2 + (1-p(v))p(v)^2 = p(v)(1-p(v))$$

and E_A becomes the expected variance of the output line for class A. □

Note that extending the above analysis to other kinds of error functions is straightforward. For example, if the error at the output is measured by $\log(1-y(v))$ when the desired output is 1, and $\log(y(v))$ when it is 0, then the terms in the sum of expected differential errors have the form

$$p(v)\log(1-y(v)) + (1-p(v))\log(y(v)).$$

Differentiating and equating to zero we again find $y(v) = p(v)$.

This short proof also strongly underlines the two conditions needed for neural networks to produce a posteriori probabilities, namely *perfect training* and *enough plasticity* of the network, so as to be able to approximate the patch of probabilities given by the lattice of differential volumes and the values $y(v)$ which we optimize independently of each other.

It is still possible to offer a simpler visual proof "without words" of the Bayesian property of classifier networks, as is done in part (b) of Figure 9.15. When training to produce 1 for the class A and 0 for A^c, we subject the function produced by the network to an "upward force" proportional to the derivative of the error function, i.e., $(1 - y(v))$, and the probability $p(v)$, and a downward force proportional to $y(v)$ and the probability $(1 - p(v))$. Both forces are in equilibrium when $p(v) = y(v)$.

This result can be visualized with the help of Figure 9.16. Several non-disjoint clusters represent different classes defined on an input space. The correspondence of each input vector to a class is given only probabilistically. Such an input space could consist for example of n-dimensional vectors, in which each component is the numerical value assigned to each of n possible symptoms. The classes defined over this input space are the different illnesses. A vector of symptoms corresponds to an affliction with some probability. This is illustrated in Figure 9.16 with the help of Gaussian-shaped probability distributions. The clusters overlap, because sometimes the same symptoms can correspond to different ailments. Such an overlap could only be suppressed by acquiring more information.

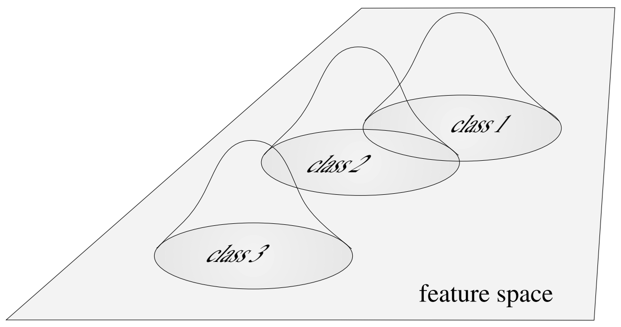


Fig. 9.16. Probability distribution of several classes in feature space

This is a nice example of the kind of application that such classification networks can have, namely in medical diagnosis. The existing data banks can be used as training sets for a network and to compute the margin of error associated with the classifications. The network can compute a first diagnosis, which is then given to a physician who decides whether or not to take this information into account.

9.3.3 Connectionist speech recognition

In automatic speech recognition we deal with the inverse problem of NETtalk: given a sequence of acoustic signals, transcribe them into text. Speech recognition is much more difficult than speech synthesis because cognitive factors play a decisive role. The recognition process is extremely sensitive to context in such a way that, if some phonemes are canceled in recorded speech, test subjects do not notice any difference. We are capable of separating speech signals from background noise (the so-called cocktail party effect) without any special effort, whereas this separation is a major computational problem for existing speech recognition systems. This leads to the suspicion that in this case deterministic rules would do much worse than a statistical system working with probabilities and likelihoods.

Building computers capable of automatically recognizing speech has been an old dream of both the field of electronics and computer science. Initial experiments were conducted as early as the 1950s, and in the 1960s some systems were already capable of recognizing vowels uttered by different speakers [12]. But until now all expectations have not been fully met. We all know of several small-scale commercial applications of speech technology for consumer electronics or for office automation. Most of these systems work with a limited vocabulary or are speaker-dependent in some way. Yet current research has as its goal the development of *large-vocabulary speaker-independent continuous speech recognition.* This long chain of adjectives already underlines the difficulties which still hamper the large-scale commercial application of automatic speech recognition: We would like the user to speak without artificial pauses, we would like that the system could understand anybody, and this without necessarily knowing the context of a conversation or monologue.

Artificial neural networks have been proposed as one of the building blocks for speech recognizers. Their function is to provide a statistical model capable of associating a vector of speech features with the probability that the vector could represent any of a given number of phonemes. Neural networks have here the function of statistical machines. Nevertheless we will see that our knowledge of the speech recognition process is still very limited so that fully connectionist models are normally not used. Researchers have become rather pragmatic and combine the best features of neural modeling with traditional algorithms or with other statistical approaches, like Hidden Markov Models, which we will briefly review. Current state-of-the-art systems combine different approaches and are therefore called *hybrid speech recognition systems.*

Feature extraction

The first problem for any automatic speech recognizer is finding an appropriate representation of the speech signal. Assume that the speech is sampled at constant intervals and denote the amplitude of the speech signal by $x(0), x(2), \ldots, x(n-1)$. For good recognition the time between consecutive measurements should be kept small. The microphone signal is thus a more or less adequate representation of speech but contains a lot of redundancy. It would be preferable to reduce the number of data points in such a way as to preserve most of the information: this is the task of all *feature extraction methods.* Choosing an appropriate method implies considering the speech production process and what kind of information is encoded in the acoustic signal.

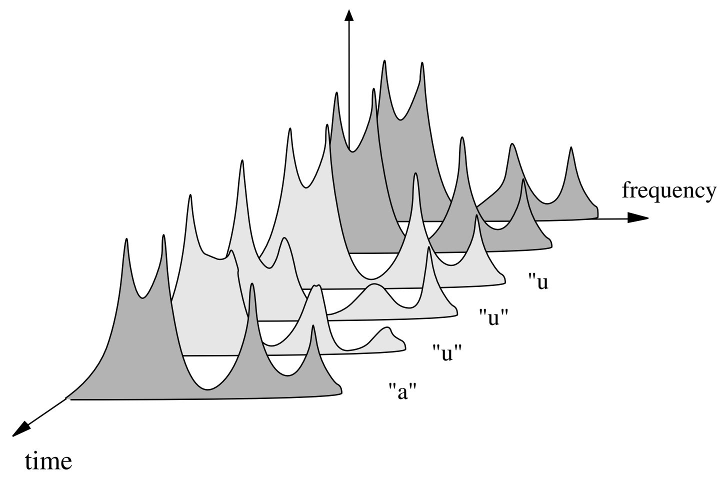


Fig. 9.17. Temporal variation of the spectrum of the speech signal

Speech is produced in the vocal tract, which can be modeled as a tube of varying diameter extending from the vocal chords to the lips. The vocal chords produce a periodic pressure wave which travels along the vocal tract until the energy it contains is released through the mouth and nose. The vocal tract behaves as a *resonator* in which some frequencies are amplified whereas others are eliminated from the final speech signal. Different configurations of the vocal organs produce different resonating frequencies, so that it is safe to assume that detecting the mixture of frequencies present in the speech signal can provide us with information about the particular configuration of the vocal tract, and from this configuration we can try to deduce what phoneme has been produced.

Figure 9.17 shows a temporal sequence of stylized spectra. The first spectrum, for example, corresponds to the vowel "a". There are four fairly distinct resonance maxima. They are called the *formants* of the speech signal. Each phoneme has a distinctive formant signature and if we could identify the sequence of formant mixtures we could, in principle, decode the speech signal.

Many methods have been proposed to deal with the task of spectral analysis of speech. Some of them have a psychophysical foundation, that is, they are based on physiological research on human hearing [353]. Others have arisen in other fields of engineering but have proved to be adequate for this task. Certainly one of the simplest, but also more powerful, approaches is computing a short-term Fourier spectrum of the speech signal.

Fourier analysis

Given a data set $\mathbf{x} = (x(0), x(2), \ldots, x(n-1))$ it is the task of Fourier analysis to reveal its periodic structure. We can think of the data set as function X

evaluated at the points $0, 2, \ldots, n-1$. The function X can be written as a linear combination of the basis functions

$$\begin{aligned}
f_0(t) &= \frac{1}{\sqrt{n}}\left(\cos\left(2\pi t\frac{0}{n}\right) - i\sin\left(2\pi t\frac{0}{n}\right)\right) = \frac{1}{\sqrt{n}}(\omega_n^*)^{0\cdot t}\\
f_1(t) &= \frac{1}{\sqrt{n}}\left(\cos\left(2\pi t\frac{1}{n}\right) - i\sin\left(2\pi t\frac{1}{n}\right)\right) = \frac{1}{\sqrt{n}}(\omega_n^*)^{1\cdot t}\\
\vdots\ &\ \ \vdots\\
f_{n-1}(t) &= \frac{1}{\sqrt{n}}\left(\cos\left(2\pi t\frac{n-1}{n}\right) - i\sin\left(2\pi t\frac{n-1}{n}\right)\right) = \frac{1}{\sqrt{n}}(\omega_n^*)^{(n-1)\cdot t}
\end{aligned}$$

where ω_n denotes the n-th complex root of unity $\omega_n = \exp(2\pi i/n)$ and ω_n^* its complex conjugate. Writing the data set as a linear combination of these functions amounts to finding which of the given frequencies are present in the data. Denote by $\mathbf{F}_n^*$ the $n \times n$ matrix whose columns are the basis functions evaluated at $t = 0, 1, \ldots, n-1$, that is, the element at row i and column j of $\mathbf{F}_n^*$ is $(\omega_n^*)^{ij}/\sqrt{n}$, for $i, j = 0, \ldots, n-1$. We are looking for a vector $\mathbf{a}$ of amplitudes such that

$$\mathbf{F}_n^*\mathbf{a} = \mathbf{x}.$$

The n-dimensional vector $\mathbf{a}$ is the *spectrum* of the speech signal. The matrix $\mathbf{F}_n$ defined as

$$\mathbf{F}_n = \frac{1}{\sqrt{n}}\begin{pmatrix}
\omega_n^0 & \omega_n^0 & \cdots & \omega_n^0\\
\omega_n^0 & \omega_n^1 & \cdots & \omega_n^{n-1}\\
\omega_n^0 & \omega_n^2 & \cdots & \omega_n^{2n-2}\\
\vdots & & \ddots & \vdots\\
\omega_n^0 & \omega_n^{n-1} & \cdots & \omega_n^{(n-1)(n-1)}
\end{pmatrix}$$

is the transpose conjugate of the matrix $\mathbf{F}_n^*$. Since the basis functions $f_0, \ldots, f_{n-1}$ are mutually orthogonal, this means that $\mathbf{F}_n^*$ is unitary and in this case

$$\mathbf{F}_n\mathbf{F}_n^*\mathbf{a} = \mathbf{F}_n\mathbf{x} \qquad \Rightarrow \qquad \mathbf{a} = \mathbf{F}_n\mathbf{x}.$$

The expression $\mathbf{F}_n\mathbf{x}$ is the discrete Fourier transform of the vector $\mathbf{x}$. The inverse Fourier transform is given of course by

$$\mathbf{F}_n^*\mathbf{F}_n\mathbf{x} = \mathbf{F}_n^*\mathbf{a} \qquad \Rightarrow \qquad \mathbf{x} = \mathbf{F}_n^*\mathbf{a}.$$

The speech signal is analyzed as follows: a window of length n (n data samples) is used to select the data. Such a window can cover, for example, 10 milliseconds of speech. The Fourier transform is computed and the magnitudes of the spectral amplitudes (the absolute values of the elements of the vector $\mathbf{a}$) are stored. The window is displaced to cover the next set of n data points and the new Fourier transform is computed. In this way we get a sequence of short-term spectra of the speech signal as a function of time, as shown in Figure 9.17. Since each articulation has a characteristic spectrum, our speech recognition algorithms should recover from this kind of information the correct sequence of phonemes.

Fast transformations

Since we are interested in analyzing the speech signal in real time it is important to reduce the number of numerical operations needed. A Fourier transform computed as a matrix-vector multiplication requires around $O(n^3)$ multiplications. A better alternative is the Fast Fourier transform, which is just a rearrangement of the matrix-vector multiplication. The left graphic in Figure 9.18 shows the real part of the elements of the Fourier matrix $\mathbf{F}_n$ (the shading is proportional to the numerical value). The recursive structure of the matrix is not immediately evident, but if the even columns are permuted to the left side of the matrix and the odd columns to the right, the new matrix structure is the one shown on the right graphic in Figure 9.18. Now the recursive structure is visible. The matrix $\mathbf{F}_n$ consists of four submatrices of dimension $n/2 \times n/2$, which are related to the matrix $\mathbf{F}_{n/2}$ through a simple formula. In order for the reduction process to work, n must be a power of two.

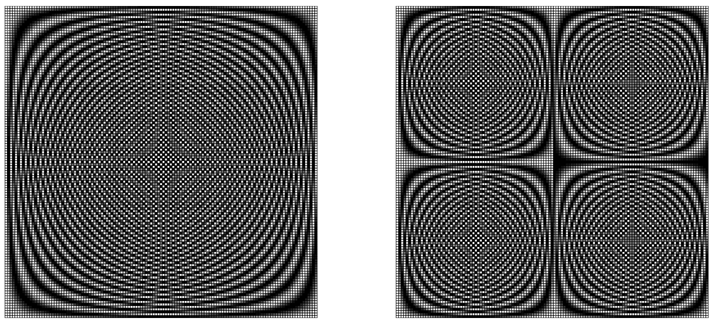


Fig. 9.18. The Fourier matrix and the permuted Fourier matrix

This rearrangement of the Fourier matrix is the basis of the Fast Fourier Transform (FFT) (See Exercise 3).

Many speech recognition systems use some kind of variation of the Fourier coefficients. The problem with short-term spectra is that the base frequency of the speaker should be separated from the medium-term information about the shape of the vocal tract. Two popular alternatives are cepstral coefficients and linear predictive coding (LPC) [353, 106].

Training of the classifier

Neural networks are used as classifier networks to compute the probability that any of a given set of phonemes could correspond to a given spectrum and the context of the spectrum. The speech signal is divided into frames of, for example, 10 ms length. For each frame the short-term spectrum or cepstrum is computed and quantized using 18 coefficients. We can train a network to associate spectra with the probability that each phoneme is present in a speech segment. A classifier network like the one shown in Figure 9.20 is used. The coefficients of the six previous and also of the six following frames are used together with the coefficients of the frame we are evaluating. The dimension of the input vector is thus 234. If we consider 61 possible phonemes we end up with the network of Figure 9.20, which is in fact very similar to NETtalk.

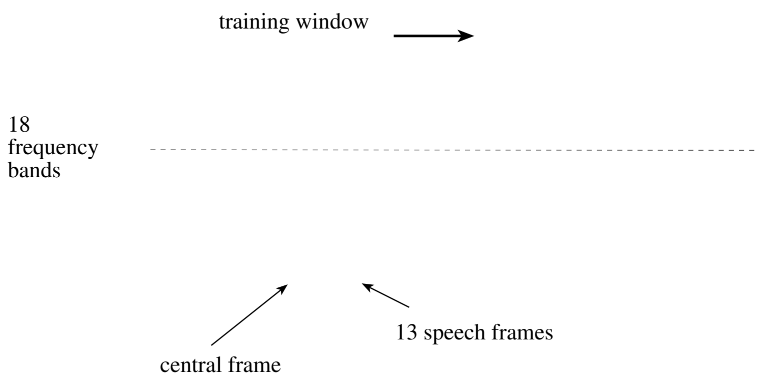


Fig. 9.19. Training window for the neural network

The network is trained with labeled speech data. There are several data bases which can be used for this purpose, but also semiautomatic methods for speech labeling can be used [65]. Once the network has been trained it can be used to compute the emission probabilities of phonemes.

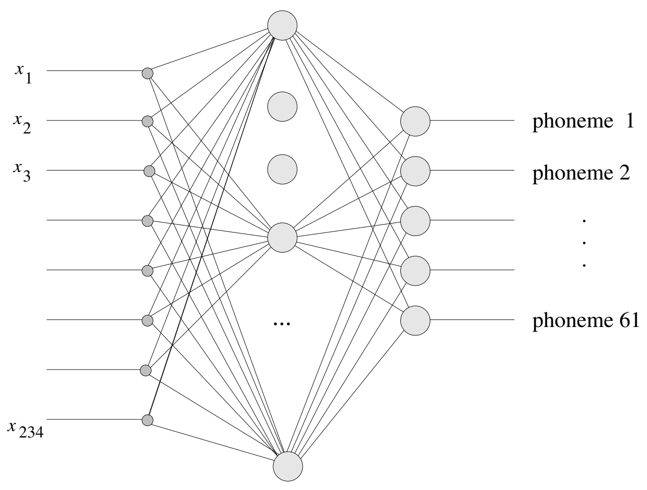


Fig. 9.20. Classification network for 61 phonemes

Hidden Markov Models

In speech recognition researchers postulate that the vocal tract shapes can be quantized into a discrete set of states roughly associated with the phonemes that compose speech. But when speech is recorded the exact transitions in the vocal tract cannot be observed and only the produced sound can be measured at some predefined time intervals. These are the *emissions*, and the *states* of the system are the quantized configurations of the vocal tract. From the measurements we want to infer the sequence of states of the vocal tract, i.e., the sequence of utterances which gave rise to the recorded sounds. In order to make this problem manageable, the set of states and the set of possible sound parameters are quantized.

A first-order Markov model is any system capable of assuming one of n different states at time t. The system does not change its state at each time step deterministically but according to a stochastic dynamic. The probability of transition from the i-th to the j-th state at each step is given by $0 \leq p_{ij} \leq 1$ and does not depend on the previous history of transitions. We also assume that at each step the model emits one of m possible output values. We call the probability of emitting the k-th output value $\mathbf{x}$ while in the i-th state $b_{ik} = P(\mathbf{x}_k|s_i)$. Starting from a definite state at time $t = 0$, the system is allowed to run for T time units and the generated outputs are recorded. Each new run of the system produces in general a different sequence of output values.

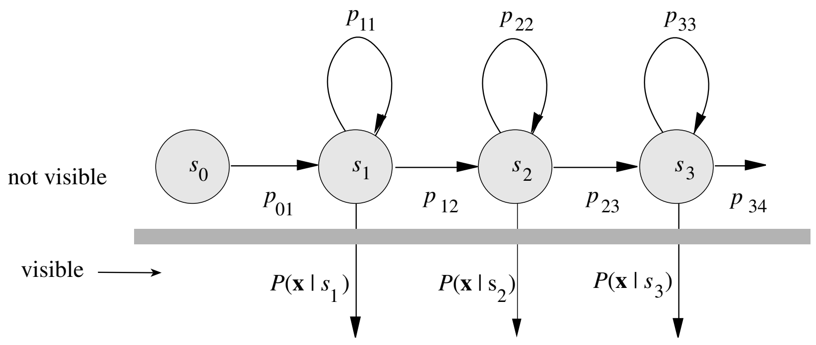


Fig. 9.21. A Hidden Markov Model

A Hidden Markov Model has the structure shown in Figure 9.21. The state transitions remain invisible for the observer (we cannot see the configuration of the vocal tract). The only data provided are the emissions (i.e., the spectrum of the signal) at some points in time. Figure 9.21 represents a model with four states, linked in such a way that we have sequential transitions. This model could represent the vocalization of a word. Each of the states s_i is a phoneme. Note that there is a probability p_{ii} that a phoneme state is repeated. This represents the case in which a speaker pronounces a word more slowly. In general, the word models constructed are more complicated than this. Especially in the case of very common words, we need more structure in the Markov model, as shown in Figure 9.22 which is a HMM for the word “and” [461]. The labeling of the nodes corresponds to the standard phonetic denomination of the relevant phonemes for this example.

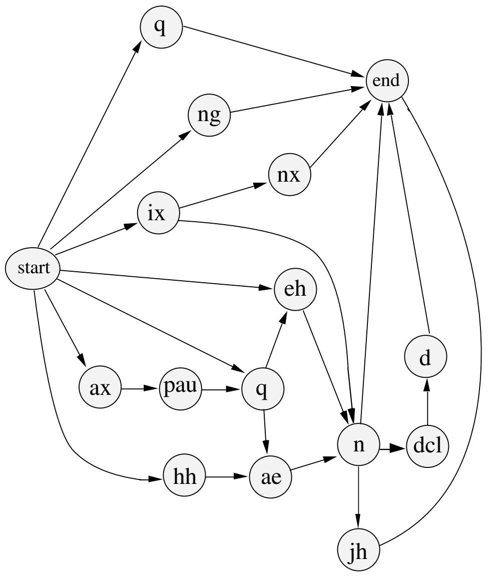


Fig. 9.22. Markov chain for the word “and”

The general problem we have when confronted with the recorded sequence of output values of a HMM is to compute the most probable sequence of state transitions which could have produced them. But first of all, we have to train the model, that is, compute the transition and emission probabilities. We discussed in Sect. 7.4.2 how this can be done applying the backpropagation algorithm.

Computation of the most probable path

Once a set of emission probabilities has been computed for several time frames $1, 2 \ldots, m$ it is necessary to compute the most probable path of transitions of the vocal tract and emissions. This can be done using dynamic programming methods of the same type as those generically known as *time warping.*

The general method is the following: the trained classifier network is applied to the speech data and for every time frame we obtain from the network the a posteriori probability of 61 phonemes. Figure 9.23 shows, for example, that for $t = 1$, that is for the first frame, the probability of having detected phoneme 1 is 0.1, for phoneme 2 it is 0.7, etc. For the second frame ($t = 2$) we get another set of 61 a posteriori probabilities and so on. We are looking for the path connecting the true sequence of produced sounds (the shaded portions of the table). The probability of any path is given by the product of the a posteriori probabilities of the phoneme sequence and the probability of transitions between phonemes. Denote by $p_a^{(t)}$ the a posteriori probability of phoneme a at time t and by $a_{i,j}$ the transition probability from phoneme i to phoneme j. Given any sequence of phonemes $k_1, k_2, \ldots, k_m$ the probability P of this special sequence is given by

$$P = p_{k_1}^{(1)} a_{k_1,k_2} p_{k_2}^{(2)} a_{k_2,k_3} \cdots p_{k_m}^{(m)} .$$

The transition probabilities are taken from the trained HMM for a word (in this case we are doing isolated word recognition). We pick the sequence of transitions with the greatest probability P and record it. The same procedure is repeated for all words in the vocabulary and the word with the greatest associated probability is selected as result of the recognition process.

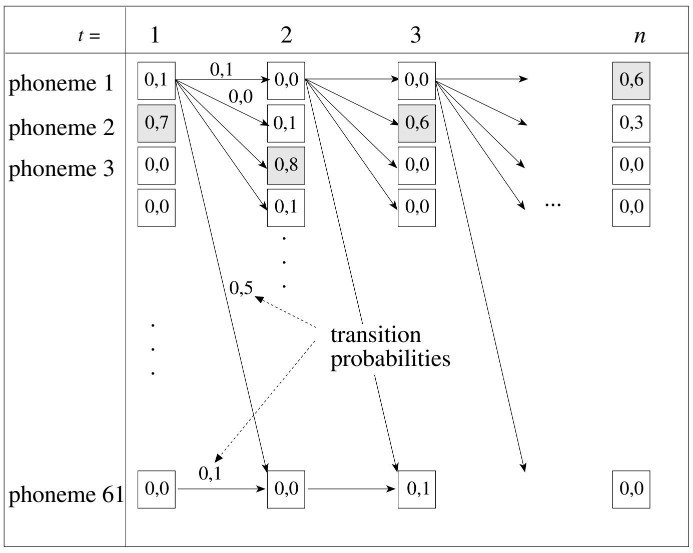


Fig. 9.23. Determination of the most probable path

The method used for the computation of the optimal path is *dynamic programming* [12]. Nevertheless it should be mentioned that one problem with this approach is that the long products of probabilities sometimes produce very small values which are difficult to discriminate. See [65] for an in-depth discussion of the pitfalls associated with speech recognition based on neural models.

9.3.4 Autoregressive models for time series analysis

It has been always an important issue to develop good forecasting techniques for time series in economics and statistics. A stochastic variable X produces a sequence of observations $x_1, x_2, \ldots, x_t$ at t different points in time that can be used to forecast the value of the variable at time $t+1$. If there is a functional relation between the successive values of the stochastic variable, we can try to formulate a linear or nonlinear model of the time series. Usually, linear models have been favored because of the accumulated experience and existing literature.

The general approach used in the neural networks field is to use historical values of the time series to train the network and test it with new values. Assume that the network has 8 input sites and one output. We can use the values $x_1, x_2, \ldots, x_8$ to forecast the value x_9. Sliding the training window one step at a time we can extract $n-8$ training examples from a time series with n data points. The network learns to forecast x_t using $x_{t-8}, \ldots, x_{t-1}$ as input. After several training steps we can measure how well the network has learned to forecast the future [291, 446].

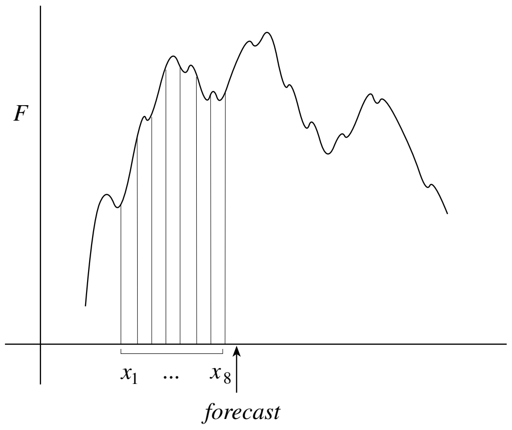


Fig. 9.24. Time series and a training window

This technique corresponds to the *autoregression models* popular among statisticians [86]. The desired approximation is of the type

$$x_t = \alpha_0 + \alpha_1 x_{t-1} + \alpha_2 x_{t-2} + \cdots + \alpha_p x_{t-p} + Z_t$$

where $\alpha_1, \ldots, \alpha_p$ are constants and Z_t a stochastic variable. This is a linear model of the kind considered before. Note that in this case we are predicting only one value into the future.

If we want to predict more than one step into the future, we can use the schema

$$x_{t+q} = \alpha_0 + (\beta_0 x_t + \cdots + \beta_{q-1} x_{t+q-1}) + (\alpha_1 x_{t-1} + \cdots + \alpha_p x_{t-p}),$$

where we use the result of previous predictions (x_t to x_{t+q-1}) in the forecast generated for step $t+q$. The system has now a built-in feedback which complicates the numerical solution. The well-known ARMA models (autoregressive moving average) have this structure.

Since nonlinear models are more general than the linear ones, we could expect that neural networks should lead to better forecasts. However, financial or other complicated time series are seldom easy to handle. Normally several statistical tests and different preprocessing techniques have to be applied before deciding on the best statistical forecasting method. In the case of economic time series, the number of degrees of freedom of the system is so large that search space has to be constrained in a decisive manner. In many cases, it is also almost impossible to base a forecast on the time series alone. If we want to forecast stock market prices, we have to consider other factors such as interest and inflation rates, foreign exchange situation, etc. It is not surprising that naive experiments where only a few parameters are considered cannot lead to successful forecasts [467].

The best results have been obtained with econometric models that relate several variables and functional dependencies. Some authors have coupled neural networks with expert systems in order to remove some of the uncertainties associated with simple-minded autoregressive models [53].

9.4 Historical and bibliographical remarks

The connection between neural networks and statistical models has spawned an active research community. Many new studies in this direction have been made since the pioneering investigations of the PDP group [421], but much work remains to be done. It should always be emphasized that feed-forward networks are a method of function approximation that must be applied carefully and with the necessary expertise from the problem domain. There are many negative examples of the kind of forecasting errors that poorly applied neural methods can produce [99]

The bootstrap was introduced by Efron [125], but similar ideas had already been proposed several years earlier in the Monte Carlo literature of hypothesis testing and by researchers trying to compute better confidence intervals. The method received its name because it models an unknown probability distribution by pulling on its own bootstraps, i.e., by resampling the given data set. A good introduction to the bootstrap method and some of its applications is [126]. Tukey [434] studied the properties of the jackknife method and gave it its name. Cross-validation has been used in many different contexts since the 1970s and some authors have investigated its usefulness in model selection.

There are now so many applications of backpropagation networks that even just mentioning the more significant would take too much space. NETtalk was one of the first examples of how a connectionist system could reach the proficiency of a rule-based system with much less effort and fewer assumptions. Other systems similar to NETtalk have been built to deal with cognitive problems like the association of visual and semantic cues. By damaging part of the network disorders such as dyslexia can be modeled. Plaut and Shallice have called this the "neurophysiology" of neural networks [345]. Another classical application is *Neurogammon*, a program that can play backgammon at the master's level. This program was the first *learning* system that could win a computer tournament. This was significant because the self-organizing neural network was able to defeat rule-based systems with a large amount of invested design work. The new version of the program, called TD-Gammon, is even better [426]. It is trained using the method of *temporal differences*, which is an especially powerful approach for nondeterministic games.

It has been an old dream of a fraction of the neural network community to apply neural networks for the forecasting of financial futures. Some banks have started projects to compare the new with the traditional methods. The published results are somewhat contradictory, because in many cases the experiments are performed off-line, that is without actual trading. Under such circumstances a high-risk approach can sometimes produce impressive results which would otherwise be forbidden under realistic conditions [245]. More disturbing is the fact that if a good forecasting technique finds its way into the real market, the whole exercise can become self-defeating. If *everybody*, or at least a significant part of the market actors, can predict the future and try to cash on this knowledge, the market will move to a new equilibrium where nobody can profit from the others. This makes the neural system of Odom and Sharda [330] the more interesting, since it can predict future bankruptcies, maybe even of one's own company. More interesting results were obtained by the networks submitted to the time series competition hosted by the Santa Fe Institute during 1992 [441]. Some neural systems were able to provide good forecasts for a wide range of time series taken from synthetic and real-world problems.

Exercises

1. Show that the mean $\bar{x}$ of n real numbers $x_1, x_2, \ldots, x_n$ is also the expected value of the mean of N bootstrap samples.
2. Train a feed-forward network to approximate a polynomial using bootstrap samples of the training set. Make a graph of the different network functions. Can you compute the confidence intervals of the functional approximation?
3. The non-symmetric discrete Fourier transform is defined using the matrix

$$\mathbf{F}_n = \begin{pmatrix} \omega_n^0 & \omega_n^0 & \cdots & \omega_n^0 \\ \omega_n^0 & \omega_n^1 & \cdots & \omega_n^{n-1} \\ \omega_n^0 & \omega_n^2 & \cdots & \omega_n^{2n-2} \\ \vdots & & \ddots & \vdots \\ \omega_n^0 & \omega_n^{n-1} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix}$$

where ω_n denotes the n-th complex root of unity (n a power of two). Show that $\mathbf{F}_n$ can be written as

$$\mathbf{F}_n = \begin{pmatrix} \mathbf{F}_{n/2} & \mathbf{D}\mathbf{F}_{n/2} \\ \mathbf{F}_{n/2} & -\mathbf{D}\mathbf{F}_{n/2} \end{pmatrix}$$

where $\mathbf{D}$ is a diagonal matrix. Derive from this result the FFT algorithm.

4. Train a network that can make a one-step prediction of a synthetic time series. Generate the data using a sum of several sinusoidal functions with different frequencies, phases, and amplitudes.

10

The Complexity of Learning

10.1 Network functions

In the previous chapters we extensively discussed the properties of multilayer neural networks and some learning algorithms. Although it is now clear that backpropagation is a statistical method for function approximation, two questions remain open: firstly, what kind of functions can be approximated using multilayer neural networks, and secondly, what is the expected computational complexity of the learning problem. We deal with both issues in this chapter.

10.1.1 Learning algorithms for multilayer networks

The backpropagation algorithm has the disadvantage that it becomes very slow in flat regions of the error function. In that case the algorithm should use a larger iteration step. However, this is precluded by the length of the gradient, which is too small in these problematic regions. Gradient descent can be slowed arbitrarily in these cases.

We may think that this kind of problem could be solved by switching the learning algorithm. Maybe there is a learning method capable of finding a solution in a number of steps that is polynomial in the number of weights in the network. But this is not so. We show in this chapter that finding the appropriate weights for a learning problem consisting of just one input-output pair is computationally hard. This means that this task belongs to the class of *NP*-complete problems, for which no polynomial time algorithm is known, because it probably does not exist.

10.1.2 Hilbert's problem and computability

The renowned mathematician David Hilbert indirectly provided the problem whose solution will bring us to the core of the function approximation issue. During his keynote speech in 1900 at the International Congress of Mathematicians in Paris, he formulated 23 problems which he identified as those whose solution would bring mathematical research forward in the 20th Century [19]. The thirteenth problem dealt with the old question of solving algebraic equations. Hilbert's hypothesis was: "It is probable that the root of an equation of seventh degree is such a function of its coefficients that it does not belong to the class of functions representable with nomographic methods, that is, it cannot be represented by a finite composition of functions of two arguments. To decide this, it should be proved that the equation

$$f^7 + xf^3 + yf^2 + zf + 1 = 0$$

of seventh degree cannot be solved using functions of two arguments".

This abstract formulation can be better understood by comparing it to similar simpler problems. Analytic formulas to solve equations of the second, third, and fourth degree, which make use exclusively of elementary arithmetic operations and of the square root function, have been known for a long time [67]. A finite number of steps is needed to get a solution. The roots of the quadratic equation $ax^2 + bx + c = 0$, for example, can be computed using a finite composition of algebraic operations, as shown in Figure 10.1.

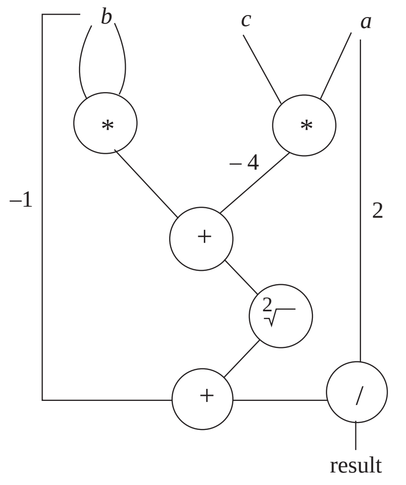


Fig. 10.1. Network for the computation of the roots of quadratic equations

Abel showed that for algebraic equations of degree higher than five, there is no such finite composition of algebraic operations that could compute its roots. Therefore there is no *algebraic formula* to find them and no finite network of algebraic nodes can be built to compute them either. However, the roots of the algebraic equation of seventh degree can be represented as a finite composition of other non-algebraic functions. A method which was very popular before the advent of computers was *nomography*, which works with graphical representations of functions. Figure 10.2 shows an example. The vertical projection of the point where the two curves $u = 2$ and $v = 1.5$ meet gives the value of $F(2, 1.5)$, where F is the function to be computed using the nomographic method. There are many functions that can be represented in this way. The composition of functions of two arguments can be computed nomographically by connecting two graphical representations using special techniques [129]. With nomography it is possible to find an approximate solution for a given equation. However, if Hilbert's hypothesis was correct, this would mean that for equations of degree higher than seven, we cannot find any finite composition of functions of two arguments to compute their roots, and therefore no nomographical composition of the kind discussed above would lead to the solution. This would be a generalization of Abel's result and would restrict the applicability of nomographic methods.

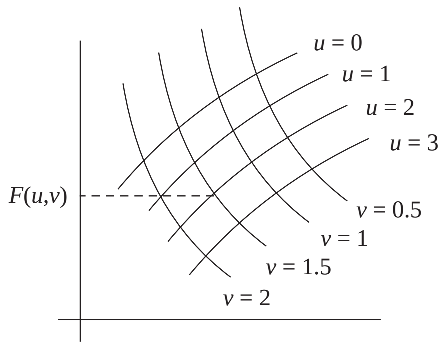


Fig. 10.2. Nomographic representation of the function $F(u, v)$

10.1.3 Kolmogorov's theorem

In 1957, the Russian mathematician Kolmogorov showed that Hilbert's conjecture does not hold. He proved that continuous functions of n arguments can always be represented using a finite composition of functions of a single argument, and addition. The theorem is certainly surprising, since it implicitly says that addition is the only function of more than one argument needed to represent continuous functions with *any* number of arguments. Multiplication, for example, can be rewritten as a composition of functions of one argument and additions, since $xy = \exp(\ln x + \ln y)$.

A modern variant of Kolmogorov's theorem, whose proof can be found in [411], states:

Proposition 14. *Let $f : [0, 1]^n \to [0, 1]$ be a continuous function. There exist functions of one argument g and ϕ_q for $q = 1, \ldots, 2n + 1$ and constants λ_p, for $p = 1, \ldots, n$ such that*

$$f(x_1, x_2, \ldots, x_n) = \sum_{q=1}^{2n+1} g\left(\sum_{p=1}^{n} \lambda_p \phi_q(x_p)\right).$$

From the perspective of networks of functions, Kolmogorov's theorem can be interpreted as stating that any continuous function of n variables can be represented by a finite network of functions of a single argument, where addition is used as the only function of several arguments. Figure 10.3 shows the kind of network needed to represent $f(x_1, x_2, \ldots, x_n)$.

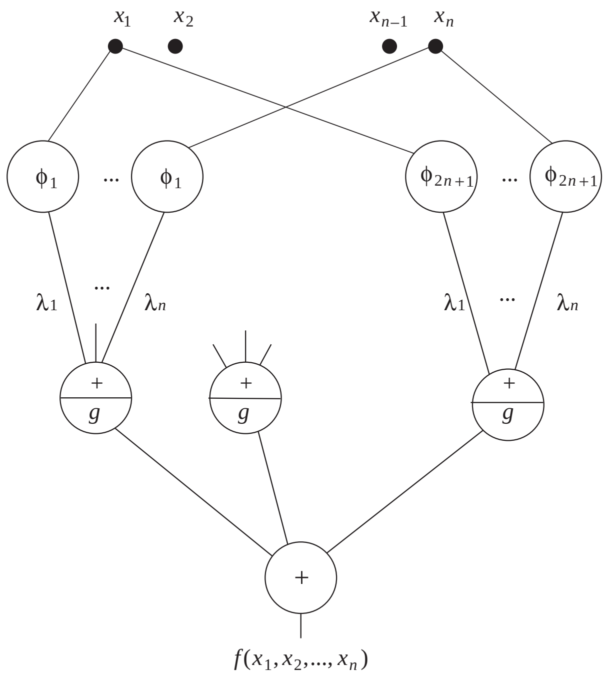


Fig. 10.3. Network for computing the continuous function f

The network is similar to those we have been using. The non-trivial functions ϕ_q can be preselected, so that only the function g and the constants $\lambda_1, \ldots, \lambda_n$ have to be found. These *Kolmogorov networks* have been compared to those that have been in use in control theory for many years [90].

Some authors have recently relaxed some of the restrictions imposed by Kolmogorov's theorem on the functions to be approximated and have studied the approximation of Lebesgue integrable functions. Irie and Miyake showed that if the hidden layer contains an unbounded number of elements, a network composed of units with fixed primitive functions at the nodes can be trained by adjusting the network weights [213]. Gallant and White have produced similar results using networks which compute Fourier series [150]. A necessary condition is that the units implement some form of *squashing function*, that is, a sigmoid or any other function of the same general type. Hornik et al. have obtained results for a broader class of primitive functions [203]. If we accept units capable of computing integral powers of the input, then we can implement polynomial approximations of a given function and Weierstrass and Stone's classical result guarantees that any real continuous function can be approximated with arbitrary precision using a finite number of computing units.

10.2 Function approximation

Kolmogorov's theorem is important in the neural networks field because it states that any continuous function can be reproduced *exactly* by a finite network of computing units, whereby the necessary primitive functions for each node *exist*. However, there is a second possibility if we want to approximate functions – we do not demand exact reproducibility but a bounded approximation error. In that case we look for the best possible approximation to a given function f. This is exactly the approach we take with backpropagation networks and any other kind of mapping networks.

10.2.1 The one-dimensional case

What kind of functions can be approximated by neural networks? To answer this question we will discuss first a more special issue. It can be shown that continuous functions $f : [0, 1] \to [0, 1]$ can be approximated with arbitrary precision. The next proposition deals with this fact, which will be extended in the following section to functions of several arguments.

Proposition 15. *A continuous real function $f : [0, 1] \to [0, 1]$ can be approximated using a network of threshold elements in such a way that the total approximation error E is lower than any given real $\varepsilon > 0$, that is,*

$$E = \int_0^1 |f(x) - \tilde{f}(x)|dx < \varepsilon,$$

where $\tilde{f}$ denotes the network function.

Proof. Divide the interval $[0, 1]$ into N equal segments selecting the points $x_0, x_1, \ldots, x_N \in [0, 1]$ with $x_0 = 0$ and $x_N = 1$. Define a function φ_N as follows:

$$\varphi_N(x) = \min\{f(x')|x' \in [x_i, x_{i+1}] \text{ for } x_i \le x \le x_{i+1}\}.$$

This function consists of N steps as shown in Figure 10.4. Now, consider φ_N an approximation of f so that the approximation error is given by

$$E_N = \int_0^1 |f(x) - \varphi_N(x)| dx.$$

Since $f(x) \geq \varphi_N(x)$ for all $x \in [0,1]$ the above integral can be written as

$$E_N = \int_0^1 f(x)dx - \int_0^1 \varphi_N(x)dx.$$

The second integral is nothing other than the lower sum of the function f in the interval $[0,1]$ with the segmentation produced by $x_0, x_1, \ldots, x_N$, as is done to define the Riemann integral. Since continuous functions are integrable, the lower sum of f converges in the limit $N \to \infty$ to the integral of f in the interval $[0,1]$. It thus holds that $E_N \to 0$ when $N \to \infty$. For any given real number $\varepsilon > 0$ there exists an M such that $E_N < \varepsilon$ for all $N \geq M$. The function φ_N is therefore the desired approximation of f. We must now show that φ_N can be computed by a neural network.

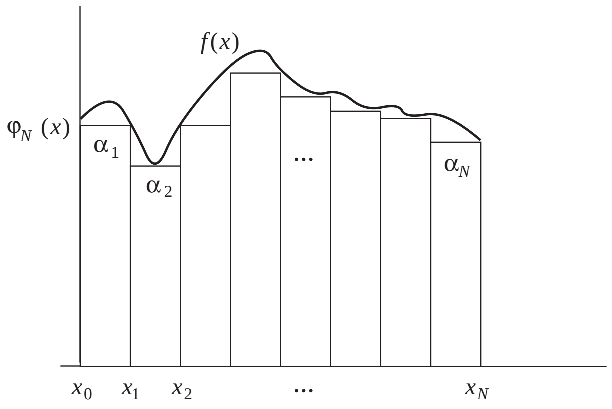


Fig. 10.4. Approximation of f with φ_N

Figure 10.4 shows the graph of φ_N for the interval $[0,1]$. The function is composed of N steps with respective heights $\alpha_1, \alpha_2, \ldots, \alpha_N$. The N segments of the interval $[0,1]$ are of the form $[x_0, x_1), [x_1, x_2), \ldots, [x_{N-1}, x_N)$.

The network shown in Figure 10.5 can compute the step-wise function φ_N. The single input to the network is x. Each pair of units with the weights x_i and x_{i+1} guarantees that the unit with threshold x_i will only fire when $x_i \leq x < x_{i+1}$. In that case the output 1 is multiplied by the weight α_{i+1}. The output unit (a linear element) adds all outputs of the upper layer of units and produces their sum as result. The unit with threshold $x_N + \delta$, where δ is positive and small, is used to recognize the case $x_{N-1} \leq x \leq x_N$.

The network shown in Figure 10.5 therefore computes the function φ_N, which approximates the function f with the desired maximal error. □

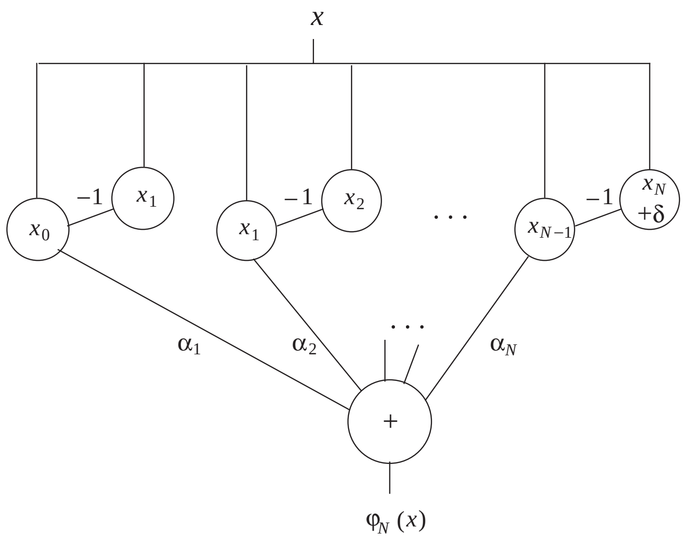


Fig. 10.5. Network for the computation of $\varphi_N(x)$

Corollary 1. *Proposition 15 is valid also for functions* $f : [0,1] \to (0,1)$ *with sigmoidal activation.*

Proof. The image of the function f has been limited to the interval $(0,1)$ in order to simplify the proof, since the sigmoid covers only this interval.

The function f can be approximated using the network in Figure 10.6. The activation of the units with threshold x_i (which is now the bias term $-x_i$) is given by $s_c(x - x_i)$, where

$$s_c(x - x_i) = \frac{1}{1 + \mathrm{e}^{-c(x-x_i)}}.$$

Different values of c produce more or less steep sigmoids. Threshold functions can be approximated with any desired precision and the network of Figure 10.6 can estimate the function φ_N with an approximation error lower than any desired positive bound.

Note that the weights for the edges connecting the first layer of units to the output unit have been set in such a way that the sigmoid produces the desired α_i values as results. It should only be guaranteed that every input x produces a single 1 from the first layer to the output unit. The first layer of the network just finds out to which of the N segments of the interval $[0,1]$ the input x belongs. $\square$

We can now generalize this results considering the case in which the function to be approximated has multiple arguments.

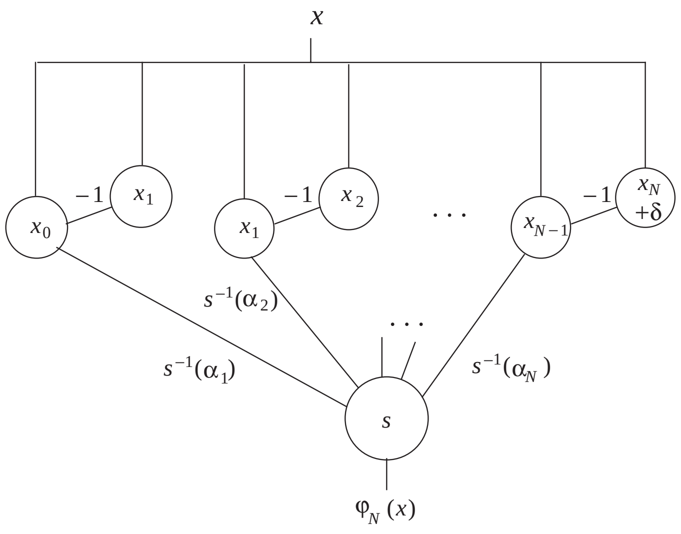


Fig. 10.6. Network for the computation of $\varphi_N(x)$ (sigmoidal units)

10.2.2 The multidimensional case

In the multidimensional case we are looking for a network capable of approximating the function $f : [0,1]^n \to (0,1)$. The network can be constructed using the same general idea as in the previous section. The approximation is computed using "blocks" as shown in Figure 10.7 for the function $\cos(x^2+y^2)$.

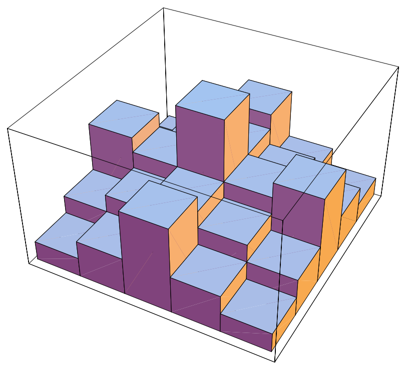


Fig. 10.7. Piecewise approximation of the function $\cos(x^2 + y^2)$

Figure 10.8 shows the necessary network extensions for a two-dimensional function. In the one-dimensional case each interval in the definition domain was recognized by two coupled units. In the two-dimensional case, it is necessary to recognize intervals in the x and y domains. This is done by using a conjunction of the outputs of the two connections as shown in Figure 10.8. The two units to the left are used to test $x_0 \leq x < x_1$. The two units to the right are used to test $y_1 \leq y < y_2$. The unit with threshold 1.5 recognizes the conjunction of both conditions. The output connection has the weight $s_0^{-1}(\alpha_{12})$, so that a unit with the sigmoid as output unit (connected afterwards) can produce the value α_{12}. This number corresponds to the desired approximation to the function f in the interval $[x_0, x_1) \times [y_1, y_2)$.

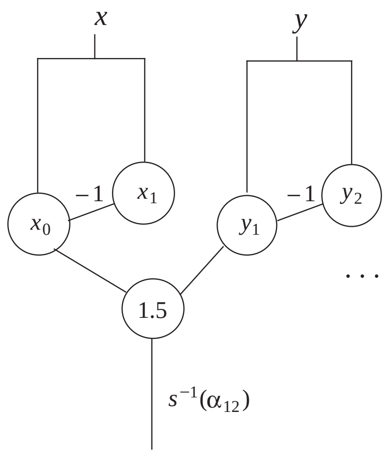


Fig. 10.8. Extension of the network for the two-dimensional case

The two-dimensional network can be completed using this scheme. Other networks for multidimensional cases ($n > 2$) can be crafted using a similar strategy. An arbitrary continuous function can be approximated using this approach but the price that has to be paid is the large number of computing units in the network. In the chapters covering self-organizing networks, we will deal with some algorithms that take care of minimizing the number of computing units needed for a good approximation.

10.3 Complexity of learning problems

Kolmogorov's theorem and Proposition 15 describe the approximation properties of networks used for the representation of functions. In the case of neural networks not only the network architecture is important but also the definition of a learning algorithm. The proof of Kolmogorov's theorem does not give any hint about the possible choice of the primitive functions for a given task, since it is non-constructive. Proposition 15 is different in this respect. Unsupervised learning can recreate something similar to the constructive approach used in the proof of the theorem.

The *general learning problem* for a neural network consists in finding the unknown elements of a given architecture. All components of the network can be partially or totally unknown, for example, the activation function of the individual units or the weights associated with the edges. However, the learning problem is usually constrained in such a way that the activation functions are chosen from a finite set of functions or some of the network weights are fixed in advance.

The *size* of the learning problem depends on the number of unknown variables which have to be determined by the learning algorithm. A network with 100 unknown weights is a much harder computational problem than a network with 10 unknown weights, and indeed much harder than the factor 1:10 would suggest. It would be helpful if the learning time could be bounded by a polynomial function in the number of variables, but this is not so.

It has been proved that the general learning problem for neural networks is *intractable*, that is, it cannot be solved efficiently for all possible instances. No algorithm is known which could solve the learning problem in polynomial time depending on the number of unknown variables. Moreover, it is very improbable that such an algorithm exists. In the terminology of complexity theory we say that *the general learning problem for neural networks is* NP-*complete*.

10.3.1 Complexity classes

Before introducing the main results concerning the complexity of learning problems, we will explain in a few pages how to interpret these results and how complexity classes are defined.

When a computational problem can be solved using a certain algorithm, the very next issue, after solvability, is the time and space complexity of the solution method. By *space complexity* we denote the memory required for the algorithmic solution once a computational model has been adopted. The *time complexity* refers to the number of algorithmic steps executed by the computational model. We are interested in the behavior of the algorithm when applied to larger and larger problems. If the index for the size of the problem is n and the number of execution steps grows asymptotically according to n^2, we speak of a quadratic time complexity of the algorithm.

Normally, space complexity is not considered a primary issue, since the computational models are provided with an infinitely large memory. What is most interesting for applications is time complexity and therefore, when we simply speak of complexity of an algorithm, we are actually referring to its time complexity.

Computational problems can be classified in a hierarchical way according to the complexity of all possible algorithms for their solution. A very common approach is to use a Turing machine as the computational model in order to help us analyze algorithms. The size n of a problem is defined as the length of the input for such a machine, when an adequate coding method has been adopted [156]. If an algorithm exists that is capable of solving *any* instance I of a problem X of size n in $p(n)$ steps, where $p(n)$ denotes a polynomial on the variable n, then X is a member of the set P of problems solvable in polynomial time. However if the solution of I requires an exponential number of steps, then X is a member of the class E of problems solvable only in exponential time. A simple example is the decoding of a binary number m as a string of m ones. Since at each step at most a single 1 can be produced as output, the number of execution steps grows exponentially with the size of the problem, which is in this case the number of bits in the binary coding for m.

However, there are some problems for which it is still unknown whether or not they belong to the class P. It has not been possible to find a polynomial time algorithm to solve any of their instances and it has also not been possible to prove that such an algorithm does not exist (which would in fact put them out of the class P). Such problems are considered *intractable*, that is, there is an inherent difficulty in dealing with them which cannot be avoided using the most ingenious methods.

The class *NP* (*nondeterministic polynomial*) has a broader definition than the class P. It contains all problems for whose solution no polynomial algorithm is known, but for which a guessed solution can be checked in polynomial time. We can explain the difference between both possibilities taking the Traveling Salesman Decision Problem (TSDP) as our benchmark. For a set of n cities in the Euclidian plane whose relative distances are known, we want to know if there is a path which visits all cities, whose total length is smaller than a given constant L. No one has been able to design an algorithm capable of solving every instance of this problem in polynomial time in the number of cities. However a proposed path can be checked in linear time: its total length is computed and we check if all cities have been visited. If more than n cities are given in the path we can reject it as invalid without additional computations. This fact, that a solution can be checked in polynomial time whereas for the problem itself no polynomial time algorithm is known, is what we mean when we say that the TSDP is a member of the class *NP*.

The class of *NP* problems derives its name from the computational model which is used for theoretical arguments and which consists of a nondeterministic computation performed in polynomial time. If a Turing machine is provided with an *oracle*, which is just an entity capable of guessing answers for a given problem, then all we need is to check the solution proposed by the oracle. The oracle works nondeterministically, in the sense that it selects one solution out of the many possible (when they exist) in a random manner. If the checking can be done in polynomial time, the problem is a member of the class *NP*. One can also think of an oracle as a parallel Turing machine which inspects all possible solutions simultaneously, eventually selecting one of the successful ones [11].

The most important open problem in complexity theory is whether the classes P and *NP* are identical or different. Theoreticians expect that it will

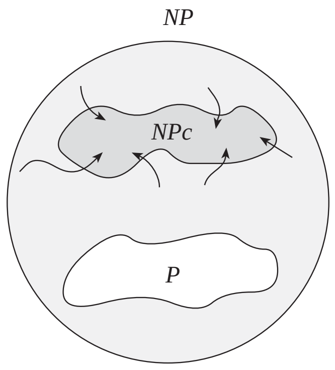


Fig. 10.9. Schematic representation of the classes *NP*, *P*, and *NPc*

eventually be possible to prove that $P \neq NP$, because otherwise a single Turing machine would be in some sense computationally equivalent to the model with an unlimited number of Turing machines. There has been some speculation that although true, the inequality $P \neq NP$ could be shown to be not provable. It has in fact been shown that in some subsets of arithmetic this is actually the case.

Obviously it is true that $P \subseteq NP$. If a problem can be solved in polynomial time, then a nondeterministic solution can also be checked in polynomial time. Just simulate the nondeterministic check by finding a solution in polynomial time. It is not known if there are some problems which belong to the class *NP* but not to the class P. Possible candidates come from the class *NPc* of so-called *NP*-complete problems.

A problem X is *NP*-complete if any other problem in *NP* can be reduced to an instance of X in polynomial time. This means that a Turing machine can be provided with a description of the original problem on its tape so that it is transformed into a description of an instance of X which is equivalent to the original problem. This reduction must be performed in polynomial time. The class *NPc* is important because if an algorithm with polynomial complexity could be found that could solve any of the problems in the class *NPc*, then any other problem in the class *NP* would be also solvable in polynomial time. It would suffice to use two Turing machines. The first one would transform the *NP* problem into an instance of the *NPc* problem with a known polynomial algorithm. The second would solve the transformed problem in polynomial time. In this sense the class *NPc* contains the most difficult problems in the class *NP*. Figure 10.9 shows a diagram of the class *NP* assuming that $NP \neq P$. The arrows in the diagram illustrate the fact that any problem in the class *NP* can be transformed into a problem in the class *NPc* in polynomial time.

Theoreticians have been able to prove that many common problems are members of the class *NPc*. One example is the *Traveling Salesman Decision Problem*, already mentioned, another the *Satisfiability Problem*, which will be discussed in the next section. The usual method used to prove that a problem A belongs to the class *NPc* is to find a polynomial transformation of a well-known problem B in *NPc* into an instance of the new problem A. This makes it possible to transform any problem in *NP* into an instance of A in polynomial time (going of course through B), which qualifies A as a member of the class *NPc*.

Proving that a problem belongs to the class *NPc* amounts to showing that the problem is computationally difficult. Theoreticians expect that no polynomial time algorithm will ever be found for those problems, that is, they expect that someday it will be possible to prove that the inequality $P \neq NP$ holds. One such computationally hard task is the general learning problem.

10.3.2 *NP*-complete learning problems

It can be shown with the help of the satisfiability problem that an *NP*-complete problem can be reduced to an instance of a learning problem for neural networks in polynomial time. The satisfiability problem is defined in the following way.

Definition 10. *Let V be a set of n logical variables, and let F be a logical expression in conjunctive normal form (conjunction of disjunctions of literals) which contains only variables from V. The satisfiability problem consists in assigning truth values to the variables in V in such a way that the expression F becomes true.*

A classical result from Cook guarantees that the satisfiability problem is a member of the class *NPc* [88]. This result also holds for such expressions F with at most three literals in each disjunction (this problem is called 3SAT in the literature). We will transform 3SAT into a learning problem for neural networks using the network shown in Figure 10.10. We give a simpler proof than Judd but keep the general outline of his method [229].

The activation of the individual units will be computed using threshold functions. An expression F in conjunctive normal form, which includes only the n variables $x_1, x_2, \ldots, x_n$ is given. The disjunctions in the normal form contain at most three literals. We want to find the truth values of the variables that make F true.

The network in Figure 10.10 has been constructed reserving for each variable x_i, a weight w_i and a computing unit with threshold 0.5. The output of the i-th unit in the first layer is interpreted as the truth value assigned to the variable x_i. The units with threshold -0.5 are used to negate the truth value of the variables x_i. The output of each of these units can be interpreted as $\neg x_i$. The third layer of units (counting from top to bottom) implements the disjunction of the outputs of the units connected to them (the clauses in the normal form). If any connection arriving to a unit in the third layer transports the value 1, the unit fires a 1. The connections in Figure 10.10 correspond to

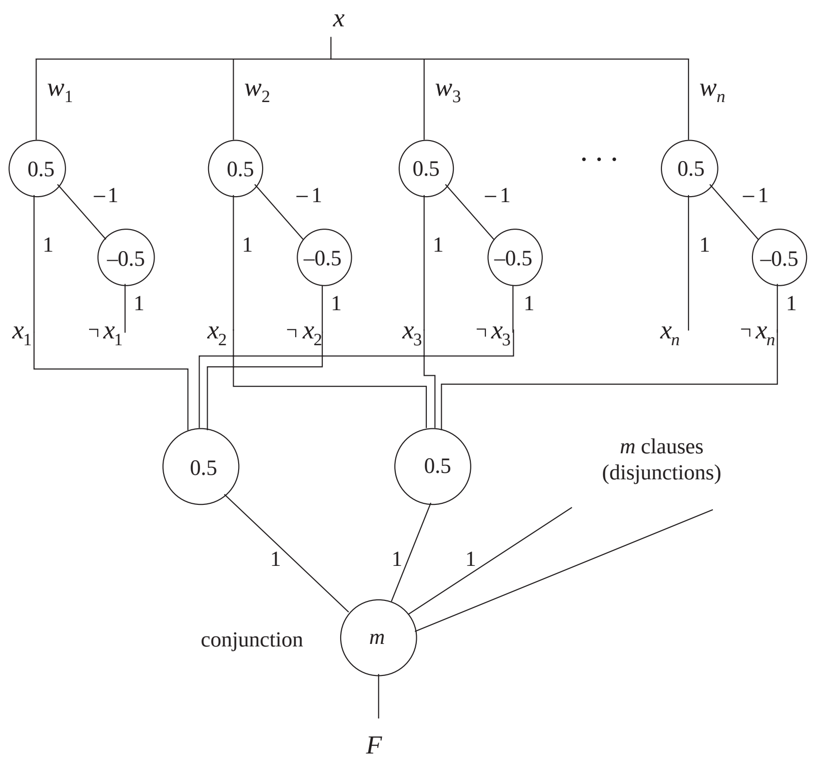


Fig. 10.10. Equivalent network for the 3SAT problem

the following two clauses: $x_1 \vee \neg x_2 \vee \neg x_3$ and $x_2 \vee x_3 \vee \neg x_n$. The last unit implements the conjunction of the disjunctions which have been hard-wired in the network. We assume that the expression F contains m clauses. The value m is used as the threshold of the single unit in the output layer. The expression F is true only if all disjunctions are true.

After this introductory explanation we can proceed to prove the following result:

Proposition 16. *The general learning problem for networks of threshold functions is* NP*-complete.*

Proof. A logical expression F in conjunctive normal form which contains n variables can be transformed in polynomial time in the description of a network of the type shown in Figure 10.10. For each variable x_i a weight w_i is defined and the connections to the units in the third layer are fixed according to the conjunctive normal form we are dealing with. This can be done in polynomial time (using a suitable coding) because it holds for the number m of different possible disjunctions in a 3SAT formula that $m \leq (2n)^3$.

After the transformation, the following learning problem is to be solved: an input $x = 1$ must produce the output $F = 1$. Only the weights in the network are to be found.

If an instantiation A with logical values of the variables x_i exists, such that F becomes true, then there exist weights $w_1, w_2, \ldots, w_n$ that solve the learning problem. It is only necessary to set $w_i = 1$ if $x_i = 1$. If $x_i = 0$ we set $w_i = 0$, that is, in both cases $w_i = x_i$. The converse is also true: if there exist weights $w_1, w_2, \ldots, w_n$ that solve the learning problem, then the instantiation $x_i = 1$, if $w_i \geq 0.5$, and $x_i = 0$ otherwise, is a valid instantiation that makes F true.

This proves that the satisfiability of logical expressions can be transformed into a learning problem for neural networks. We must now show that the learning problem belongs in the class *NP*, that is, that a solution can be checked in polynomial time.

If the weights $w_1, w_2, \ldots, w_n$ are given, then a single run of the network can be used to check if the output F is equal to 1. The number of computation steps is directly proportional to the number n of variables and to the number m of disjunctive clauses, which is bounded by a polynomial in n. The time required to check an instantiation is therefore bounded by a polynomial in n. This means that the given learning problem belongs to the class *NP*. □

It could be argued that the learning problem stated above is more difficult to solve than in the typical case, because many of the network weights have been selected and fixed in advance. Usually all weights and thresholds are considered as variables. One could think that if all weights are set free to vary, this opens more regions of weight space for exploration by the learning algorithm and that this could reduce the time complexity of the learning problem. However this is not so. Proposition 16 is still valid even when all weights are set free to be modified. The same is true if the threshold function is substituted by a sigmoid [229]. It has even been shown that the training of a three-unit network can be *NP*-complete for a certain training set [324].

10.3.3 Complexity of learning with AND-OR networks

Since the general learning problem is *NP*-complete, we can try to analyze some kinds of restricted architectures to find out if they can be trained in polynomial time.

As a second example we consider networks of units which can only compute the AND or the OR function. This restriction limits the number of available combinations and we could speculate that this helps to avoid any combinatorial explosion. However, this is not the case, and it can be shown that the learning problem for this kind of network remains *NP*-complete.

In the proof of Proposition 16 we constructed a network which mirrored closely the satisfiability problem for logical expressions. The main idea was to produce two complementary output values for each logical variable, which were identified as x_i and $\neg x_i$. These outputs were connected to disjunction units, and the output of the disjunctions was connected to the output unit (which computed the AND function). Any logical expression can be hardwired in such way. For the proof of the following proposition we will use the same approach, but each unit will be more flexible than before.

Proposition 17. *The learning problem for neural network architectures whose units can only compute the AND or the OR function is* NP*-complete.*

Proof. Just as we did in the proof of Proposition 16, we will show that a polynomial time learning algorithm for the networks considered could also solve the 3SAT problem in polynomial time. A logical expression F in conjunctive normal form, with at most three literals in each disjunction, will be computed by the network shown in Figure 10.11.

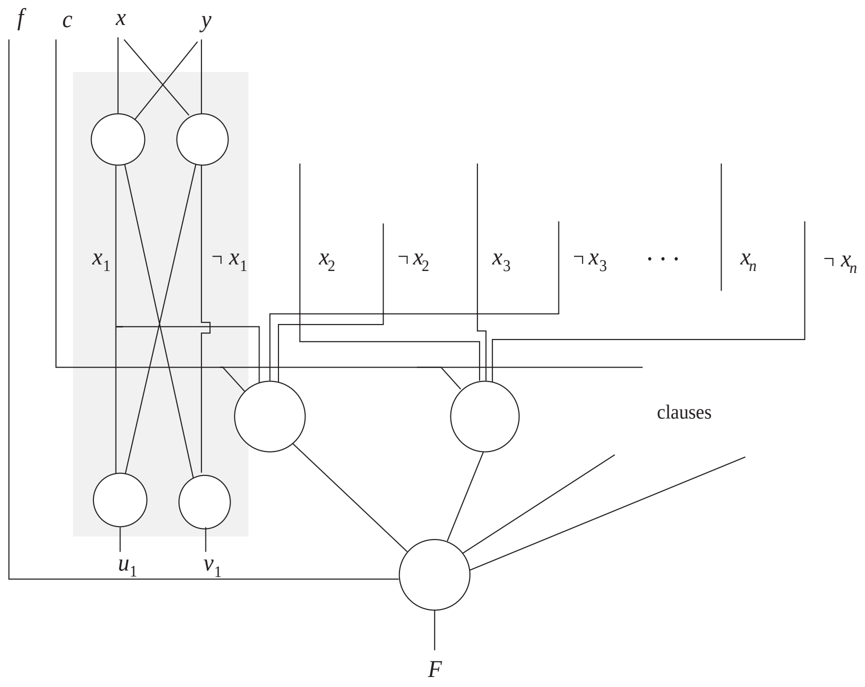


Fig. 10.11. Equivalent network for the 3SAT problem with AND-OR units

The arrangement is similar to the one in Figure 10.10, but with some differences. There are now four inputs f, c, x, y. The values x and y are connected to each column of units in the network (to avoid cluttering the diagram we show only one module for each variable). The shaded module is present n times in the network, that is, once for each logical variable in the expression F. The outputs of each module (x_i and $\neg x_i$) are shown in the diagram.

The weights of the network are unknown, as are the thresholds of each unit (whose activation function is a simple threshold function). Only those combinations of weights and thresholds are allowed that lead to the computation of the functions AND and OR at the nodes. The learning problem for the network is given by the following table, which consists of three input-output pairs:

	x	y	f	c	F	u_1	v_1	u_2	v_2	$\cdots$	u_n	v_n
(a)	0	0	1	1	1	0	0	0	0	$\cdots$	0	0
(b)	0	0	1	0	0	0	0	0	0	$\cdots$	0	0
(c)	0	1	1	1	1	0	1	0	1	$\cdots$	0	1

Since the individual units only compute AND or OR functions, this means that the input $x = y = 0$ makes all output lines $x_1, \neg x_1, x_2, \neg x_2, \ldots, x_n, \neg x_n$ equal to 0. From row (b) of the training set table, we know that $F = 0$. Since the only input to the output unit are the disjunctions of the literals (which are all equal to 0) and $f = 1$, this means that the output unit must compute the AND function. Row (b) therefore determines the kind of output unit that we can use.

Row (a) of the learning problem tells us that the "clause" units, to which the values x_i and $\neg x_i$ are hard-wired, must compute the OR function. This is so because in row (a) $F = 1$ and $c = 1$. Note that c is connected to all the clause units and that all literals are zero (since, again, $x = y = 0$). The only possibility to get $F = 1$ is for all clause units to produce a 1, so that the AND output unit can also produce a 1. This can only happen if the clause units are disjunctions.

Rows (a) and (b) together force the network to compute a conjunctive normal form. Since we have taken care to connect the literals x_i and $\neg x_i$ in the order required by expression F, we have a similar situation to the one we had in Figure 10.10.

Row (c) of the learning problem provides us with the final argument to close the proof. Now we have $x = 0$ and $y = 1$. Since we want the values of u_i and v_i to be complementary, as row (c) demands, this can only happen if x_i and $\neg x_i$ are themselves complementary. Otherwise u_i would be always equal to v_i, because the AND and the OR function produce the same outputs for the input combinations (0,0) and (1,1). The connections x_i and $\neg x_i$ are thus in fact complementary.

If the learning problem has a solution, the values of the connections $x_1, x_2, \ldots, x_n$ provide the instantiation of the logical variables that we are looking for to satisfy the expression F. The additional details of the proof can be worked out as in the proof of Proposition 16. $\square$

Judd has also proved a more general theorem. Even if all linear threshold functions can be selected as activation functions, the learning problem remains *NP*-complete [229].

10.3.4 Simplifications of the network architecture

The complexity results of the last section show that training neural networks is a computationally hard problem. If the architecture of the network is only loosely defined and if the training algorithm must find a valid configuration in a large region of weight space, then no polynomial time algorithm can be used.

This situation makes it necessary to improve the known training methods for a fixed size of the learning problem. We discussed this when we analyzed fast variations of backpropagation in Chap. 8. A more promising approach for breaking the "curse of dimensionality", brought by the combinatorial explosion of possible weight combinations, is to test different kinds of simplified architectures. Judd has analyzed some kinds of "flat" neural networks to try to discover if, at least for some of them, polynomial time learning algorithms can be designed.

The motivation behind the theoretical analysis of flat neural networks is the structure of the human cortex. The cortex resembles a two-dimensional structure of low depth. It could be that this kind of architecture can be trained more easily than fully connected networks. However, Judd could show that learning in flat two-dimensional networks is also *NP*-complete [228]. Another kind of network, rather "unbiological" one-dimensional strings of low depth, can be trained in linear time. And if only the average case is considered (and not the worst-case, as usually done in complexity arguments) the training time can even be reduced to a constant, if a parallel algorithm is used.

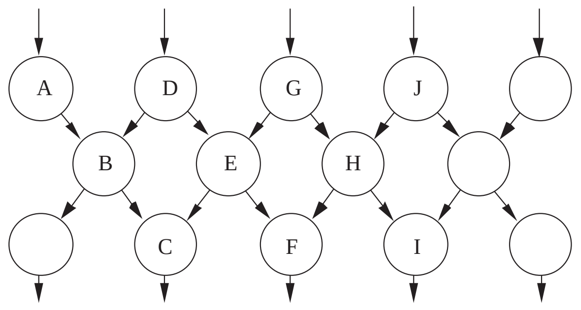


Fig. 10.12. A network of three layers in one direction

Figure 10.12 shows a "one-dimensional" network. The inputs come from the top and the output is produced by the third layer of units. The interconnections are defined in such a way that each unit propagates signals only to its neighbors to the left and to the right. No signal is propagated over more than three stages. The i-th bit of the input can affect at most three bits of the output (Figure 10.12).

Flat three-stage networks can be built out of modules that determine the output of a unit. The output of unit C, for example, depends only on information from the units A, B, D, E, and G. These units constitute, together with C, an *information column*. The units D, E, F, G, H, and J constitute the neighboring column. Both structures overlap and it is necessary to find a correct combination for a given learning problem. Assume that the input and the output is binary and that the units in the network are threshold elements of two inputs (which can compute 14 different logical functions). Each information column made of 6 units can assume any of 14^6 configurations. Not all of them are compatible with the configuration of the neighboring column. The combinatorial problem can be solved in two steps: firstly, the valid configurations of the columns are determined using the given input and output vectors. Secondly, only those configurations compatible with those of the neighbors are accepted. In a one-dimensional network with n output bits we must combine the output of n different columns. Judd has shown that this computation can be performed in linear time [229]. In another paper he took the next step and assumed that a processor is available to train each column and considered the average case [230]. Some computer experiments show that in the average case, that is, when the input-output pairs are selected randomly, the time required for coordination of the information columns is bounded by a constant. The network can be trained in constant time independently of its width.

Although the one-dimensional case cannot be considered very realistic, it gives us a hint about a possible strategy for reducing the complexity of the learning problem: if the network can be modularized and the information exchange between modules can be limited in some way, there is a good chance that an efficient learning algorithm can be found for such networks. Some of these ideas have been integrated recently into models of the human cortex [78]. The biological cortex has a flat, two-dimensional structure with a limited number of cortical layers and a modular structure of the cortical columns [360]. In Burnod's model the *cortical columns* are the next hierarchical module after the neurons, and are also learning and functional substructures wired as a kind of cellular automata. It seems therefore that modularization of the network structures, in biological organisms as well as in artificial neural networks, is a necessary precondition for the existence of efficient learning algorithms.

10.3.5 Learning with hints

The example of the one-dimensional networks illustrates one method capable of stopping the combinatorial explosion generated by learning problems. Another technique consists in considering some of the properties of the function to be modeled before selecting the desired network architecture. The number of degrees of freedom of the learning problem can be reduced exploiting "hints" about the shape or the properties of the function to be approximated.

Assume that we want to approximate a real *even* function f of n arguments using a three-layered network. The network has n input sites and a single output unit. The primitive function at the units is the symmetric sigmoid, i.e., an odd function. We can force the network to compute exclusively even functions, that is those for which it holds that $\varphi(x_1, x_2, \ldots, x_n) = \varphi(-x_1, -x_2, \ldots, -x_n)$. This can be guaranteed by replicating each unit and each weight in the second layer. Figure 10.13 shows how to interconnect the units in the network. Both inputs, x_i or $-x_i$, lead to the same output of the network. If the whole network is hardwired following this approach (for each single variable), it will only be capable of computing even functions. This produces a reduction of the search region in the space of functions [5].

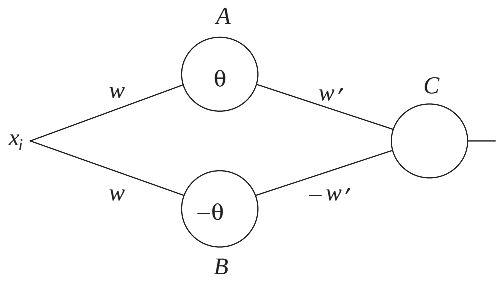


Fig. 10.13. Network to compute even functions

This is the general method to include hints or conditions in a network. If backpropagation is used as the learning method we must also take care to keep the identity of all duplicated weights in the network. We already saw in Chap. 7 how this can be done. All corrections to a weight with the same name are computed independently, but are added before making the weight update. In the case where the weights have the same name but a different sign, the weight corrections are multiplied by the corresponding sign before being added. In this way we keep the original structure of the network, and weights which should be identical remain so (up to the sign).

Some other kinds of invariance are more difficult to implement. Duplicating the network and comparing the respective output values helps to do this. Assume that a network must produce the same real results when two different n-dimensional input values $\mathbf{x}_1$ and $\mathbf{x}_2$ are presented. The difference between the two vectors can reflect some kind of invariance we want to enforce. The vector $\mathbf{x}_2$ could be equal to $-\mathbf{x}_1$ or the components of $\mathbf{x}_2$ could be some permutation of the components of $\mathbf{x}_1$. We would like to compute a function invariant under such modifications of the input.

Figure 10.14 shows the general strategy used to solve this problem. A single untrained network produces the output y_1 for the input $\mathbf{x}_1$ and the output y_2 for the input $\mathbf{x}_2$. The network can be forced to bring y_1 and y_2 closer by minimizing the function $(y_1 - y_2)^2$. This is done by duplicating the network and by performing the additional computation $(y_1 - y_2)^2/2$. Since we want the network to produce a target value t, the difference between y_2 and t must also be minimized. The right side of the additional computing units shown in Figure 10.14 shows the difference that must be minimized, whereas the left side contains the derivative according to y_2. The extended network of Figure 10.14 can be trained with standard backpropagation.

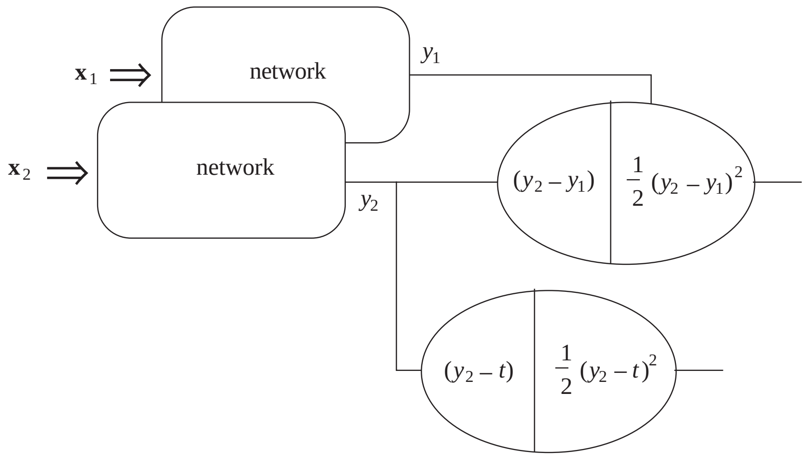


Fig. 10.14. Extended network to deal with invariants

The learning algorithm must take care of duplicated weights in the network as was discussed in Chap. 7. Using gradient descent we try to force the network to produce the correct result and to respect the desired invariance. An alternative approach is to specialize the first layers of the network to the production of the desired invariance and the last ones to the computation of the desired output.

Hints, that is, knowledge about necessary or desired invariances of the network function, can also be used to preprocess the training set. If the network function must (once again) be even, we can expand the training set by introducing additional input-output pairs. The new input data is obtained from the old just by changing the sign, whereas the value of the target outputs remains constant. A larger training set reduces the feasible region of weight space in which a solution to the learning problem can be found. This can lead to a better approximation of the unknown function if the VC dimension of the search space is finite [6]. The technique of "learning with hints" tries to reduce the inherent complexity of the learning problem by reducing the degrees of freedom of the neural network.

10.4 Historical and bibliographical remarks

A. N. Kolmogorov's theorem was rediscovered in the 1980s by Hecht-Nielsen [1987b] and applied to networks of functions. He relied heavily on the work of Lorentz]1976], who had described the full scope of this theorem some years earlier. Since then many different types of function networks and their properties have been investigated, so that a lot of results about the approximation properties of networks are already available.

Minsky and Papert [1969] had already made some complexity estimates about the convergence speed of learning algorithms. However, at that time it was not possible to formulate more general results, as the various classes of complexity were not defined until the 1970s in the work of Cook [1971] and others. Karp [1972] showed that a broad class of problems is *NP*-complete. In his Turing Award Lecture he described the intellectual roots of this complexity research.

In his dissertation Judd [1990] presented the most important theorems on the complexity of learning algorithms for neural networks. His results have been extended over the last few years. It has been shown that when the number of degrees of freedom and combinational possibilities of evaluation elements goes beyond a certain threshold, the learning problems become *NP*-complete. Parberry has collected many interesting results and has helped to systematize the study of complexity of neural networks [335].

The efforts of authors such as Abu-Mostafa, Judd, and others to reduce the complexity of the learning problem are reflected in the research of authors such as Kanerva [1992], who have set up simpler models of biological networks in order to explain their great ability to adapt and their regular column structure.

The existence theorems in this chapter and the complexity estimations made for the learning problem may appear superfluous to the practitioner. However, in a field in which numerical methods are used intensively, we must be aware of the limits of efficient computations. Only on this basis can we design network architectures that will make faster solutions to learning problems possible.

Exercises

1. Compute the approximation error of a sigmoid to the step function for different values of the constant c, where $s(x) = 1/(1 + \exp(-cx))$. What is the approximation error to a function approximated by the network of Figure 10.6?
2. Rewrite the function $((xy/z) - x)/y$ using only addition and functions with one real argument.
3. Prove that the network of Figure 10.13 computes an even function.
4. Propose a network architecture and a learning method to map a set of points in $\mathbb{R}^5$ to a set of points in $\mathbb{R}^2$, in such a way that points close to each other in $\mathbb{R}^5$ map as well as possible to neighboring points in $\mathbb{R}^2$.

11

Fuzzy Logic

11.1 Fuzzy sets and fuzzy logic

We showed in the last chapter that the learning problem is *NP*-complete for a broad class of neural networks. Learning algorithms may require an exponential number of iterations with respect to the number of weights until a solution to a learning task is found. A second important point is that in backpropagation networks, the individual units perform computations more general than simple threshold logic. Since the output of the units is not limited to the values 0 and 1, giving an interpretation of the computation performed by the network is not so easy. The network acts like a black box by computing a statistically sound approximation to a function known only from a training set. In many applications an interpretation of the output is necessary or desirable. In all such cases the methods of *fuzzy logic* can be used.

11.1.1 Imprecise data and imprecise rules

Fuzzy logic can be conceptualized as a generalization of classical logic. Modern fuzzy logic was developed by Lotfi Zadeh in the mid-1960s to model those problems in which imprecise data must be used or in which the rules of inference are formulated in a very general way making use of diffuse categories [170]. In fuzzy logic, which is also sometimes called diffuse logic, there are not just two alternatives but a whole continuum of truth values for logical propositions. A proposition A can have the truth value 0.4 and its complement A^c the truth value 0.5. According to the type of negation operator that is used, the two truth values must not be necessarily add up to 1.

Fuzzy logic has a weak connection to probability theory. Probabilistic methods that deal with imprecise knowledge are formulated in the Bayesian framework [327], but fuzzy logic does not need to be justified using a probabilistic approach. The common route is to generalize the findings of multivalued logic in such a way as to preserve part of the algebraic structure [62]. In this chapter we will show that there is a strong link between set theory, logic, and geometry. A fuzzy set theory corresponds to fuzzy logic and the semantic of fuzzy operators can be understood using a geometric model. The geometric visualization of fuzzy logic will give us a hint as to the possible connection with neural networks.

Fuzzy logic can be used as an interpretation model for the properties of neural networks, as well as for giving a more precise description of their performance. We will show that fuzzy operators can be conceived as generalized output functions of computing units. Fuzzy logic can also be used to specify networks directly without having to apply a learning algorithm. An expert in a certain field can sometimes produce a simple set of control rules for a dynamical system with less effort than the work involved in training a neural network. A classical example proposed by Zadeh to the neural network community is developing a system to park a car. It is straightforward to formulate a set of fuzzy rules for this task, but it is not immediately obvious how to build a network to do the same nor how to train it. Fuzzy logic is now being used in many products of industrial and consumer electronics for which a *good* control system is sufficient and where the question of *optimal* control does not necessarily arise.

11.1.2 The fuzzy set concept

The difference between crisp (i.e., classical) and fuzzy sets is established by introducing a *membership function*. Consider a finite set $X = \{x_1, x_2, \ldots, x_n\}$ which will be considered the universal set in what follows. The subset A of X consisting of the single element x_1 can be described by the n-dimensional membership vector $Z(A) = (1, 0, 0, \ldots, 0)$, where the convention has been adopted that a 1 at the i-th position indicates that x_i belongs to A. The set B composed of the elements x_1 and x_n is described by the vector $Z(B) = (1, 0, 0, ..., 1)$. Any other crisp subset of X can be represented in the same way by an n-dimensional binary vector. But what happens if we lift the restriction to binary vectors? In that case we can define the *fuzzy set* C with the following vector description:

$$Z(C) = (0.5, 0, 0, ..., 0)$$

In classical set theory such a set cannot be defined. An element belongs to a subset or it does not. In the theory of fuzzy sets we make a generalization and allow descriptions of this type. In our example the element x_1 belongs to the set C only to some extent. The degree of membership is expressed by a real number in the interval $[0, 1]$, in this case 0.5. This interpretation of the degree of membership is similar to the meaning we assign to statements such as “person x_1 is an adult”. Obviously, it is not possible to define a definite age which represents the absolute threshold to enter into adulthood. The act of becoming mature can be interpreted as a continuous process in which the membership of a person to the set of adults goes slowly from 0 to 1.

There are many other examples of such diffuse statements. The concepts “old” and “young” or the adjectives “fast” and “slow” are imprecise but easy to interpret in a given context. In some applications, such as expert systems, for example, it is necessary to introduce formal methods capable of dealing with such expressions so that a computer using rigid Boolean logic can still process them. This is what the theory of fuzzy sets and fuzzy logic tries to accomplish.

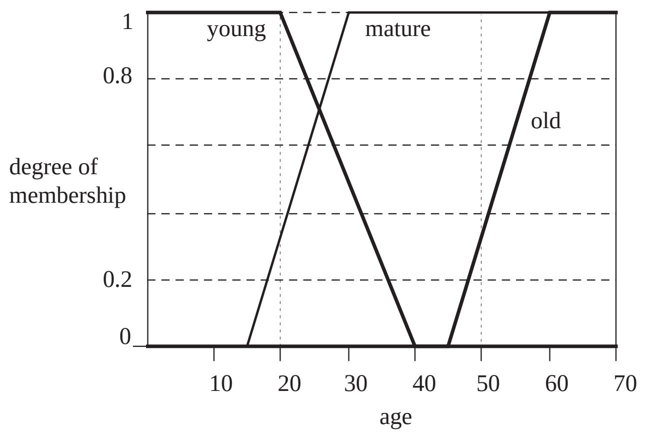


Fig. 11.1. Membership functions for the concepts young, mature and old

Figure 11.1 shows three examples of a membership function in the interval 0 to 70 years. The three functions define the degree of membership of any given age in the sets of young, adult, and old ages. If someone is 20 years old, for example, his degree of membership in the set of young persons is 1.0, in the set of adults 0.35, and in the set of old persons 0.0. If someone is 50 years old the degrees of membership are 0.0, 1.0, 0.3 in the respective sets.

Definition 11. *Let X be a classical universal set. A real function $\mu_A : X \to [0,1]$ is called the membership function of A and defines the fuzzy set A of X. This is the set of all pairs $(x, \mu_A(x))$ with $x \in X$.*

A fuzzy set is completely determined by its membership function. Note that the above definition also covers the case in which X is not a finite set.

The *set of support* of a fuzzy set A is the set of all elements x of X for which $(x, \mu_A(x)) \in A$ and $\mu_A(x) > 0$ holds. A fuzzy set A with the finite set of support $\{a_1, a_2, \ldots, a_m\}$ can be described in the following way

$$A = \mu_1/a_1 + \mu_2/a_2 + \cdots + \mu_m/a_m,$$

where $\mu_i = \mu_A(a_i)$ for $i = 1, \ldots, m$. The symbols “/” and “+” are used only as syntactical constructors.

Crisp sets are a special case of fuzzy sets, since the range of the function is restricted to the values 0 and 1. Operations defined over crisp sets, such as union or intersection, can be generalized to cover also fuzzy sets.

Assume as an example that $X = \{x_1, x_2, x_3\}$. The classical subsets $A = \{x_1, x_2\}$ and $B = \{x_2, x_3\}$ can be represented as

$$A = 1/x_1 + 1/x_2 + 0/x_3 \qquad B = 0/x_1 + 1/x_2 + 1/x_3.$$

The union of A and B is computed by taking for each element x_i the maximum of its membership in both sets, that is:

$$A \cup B = 1/x_1 + 1/x_2 + 1/x_3$$

The *fuzzy union* of two fuzzy sets can be computed in the same way. The union of the two fuzzy sets

$$C = 0.5/x_1 + 0.6/x_2 + 0.3/x_3 \qquad D = 0.7/x_1 + 0.2/x_2 + 0.8/x_3$$

is given by

$$C \cup D = 0.7/x_1 + 0.6/x_2 + 0.8/x_3$$

The fuzzy intersection of two sets A and B can be defined in a similar way, but instead of taking the maximum we compute the minimum of the membership of each element x_i to A and B. The maximum or minimum of the membership values are just one pair of possible definitions of the union and intersection operations for fuzzy sets. As we show later on, there are other alternative definitions.

11.1.3 Geometric representation of fuzzy sets

Bart Kosko introduced a very useful graphical representation of fuzzy sets [259]. Figure 11.2 shows an example in which the universal set consists only of the two elements x_1 and x_2. Each point in the interior of the unit square represents a subset of X. The convention is that the coordinates of the representation correspond to the membership values of the elements in the fuzzy set. The point $(1,1)$, for example, represents the universal set X, with membership function $\mu_A(x_1) = 1$ and $\mu_A(x_2) = 1$. The point $(1,0)$ represents the set $\{x_1\}$ and the point $(0,1)$ the set $\{x_2\}$. The crisp subsets of X are located at the vertices of the unit square. The geometric visualization can be extended to an n-dimensional hypercube.

Kosko calls the inner region of a unit hypercube in an n-dimensional space the *fuzzy region*. We find here all combinations of membership values that a fuzzy set could assume. The point M in Figure 11.2 corresponds to the fuzzy set $M = 0.5/x_1 + 0.3/x_2$. The center of the square represents the most diffuse of all possible fuzzy sets of X, that is the set $Y = 0.5/x_1 + 0.5/x_2$.

The degree of fuzziness of a fuzzy set can be measured by its *entropy*. In the geometric visualization, this corresponds inversely to the distance between

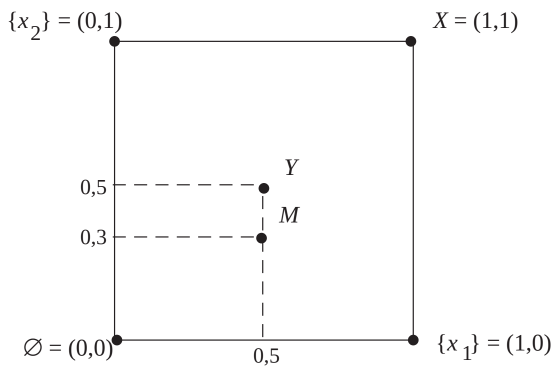


Fig. 11.2. Geometric visualization of fuzzy sets

the representation of the set and the center of the unit square. The set Y in Figure 11.3 has the maximum possible entropy. The vertices represent the crisp sets and have the lowest entropy, that is, zero. Note that the fuzzy concept of entropy is mathematically different from the entropy concept in physics or information theory. Some authors prefer to use terms like *index of fuzziness* [239] or also *crispness*, *certitude*, *ambiguity*, etc. [55].

With this caveat we adopt a preliminary definition of the entropy of a fuzzy set M as the quotient of the distance d_1 (according to some metric) of the corner which is nearest to the representation of M to the distance d_2 from the corner which is farthest away. Figure 11.3 shows the two relevant segments. The entropy $E(M)$ of M is therefore

$$E(M) = \frac{d_1}{d_2}.$$

According to this definition the entropy is bounded by 0 and 1. The maximum entropy is reached at the center of the square.

The union or intersection of sets can be also visualized using this representation. The membership function for the the union of two sets A and B can be defined as

$$\mu_{A\cup B}(x) = \max(\mu_A(x), \mu_B(x)) \quad \forall x \in X \tag{11.1}$$

and corresponds to the maximum of the corresponding coordinates in the geometric visualization. The membership function for the intersection of two sets A and B is given by

$$\mu_{A\cap B}(x) = \min(\mu_A(x), \mu_B(x)) \quad \forall x \in X. \tag{11.2}$$

Together with the points representing the sets A and B, Figure 11.4 shows the points which represent their union and intersection.

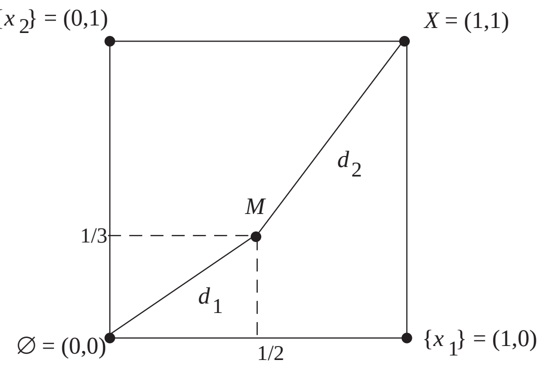


Fig. 11.3. Distance of the set M to the universal and to the void set

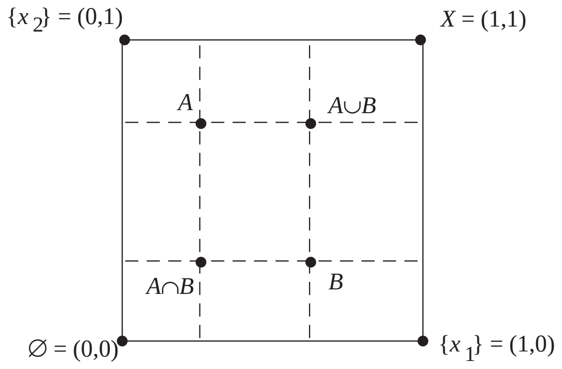


Fig. 11.4. Intersection and union of two fuzzy sets

The union or intersection of two fuzzy sets is in general a fuzzy, not a crisp set. The complement A^c of a fuzzy set A can be defined with the help of the membership function μ_{A^c} given by

$$\mu_{A^c}(x) = 1 - \mu_A(x) \quad \forall x \in X\,. \tag{11.3}$$

Figure 11.5 shows that the representation of A must be transformed into another point at the same distance from the center of the unit square. The line joining the representation of A and A^c goes through the center of the square. Figure 11.5 also shows how to obtain the representations for $A \cup A^c$ and $A \cap A^c$ using the union and intersection operators defined before. For fuzzy sets, it holds in general that

$$A \cup A^c \neq X \qquad \text{and} \qquad A \cap A^c \neq \emptyset$$

which is not true in classical set theory. This means that the principle of excluded middle and absence of contradiction do not necessarily hold in fuzzy logic. The consequences will be discussed later.

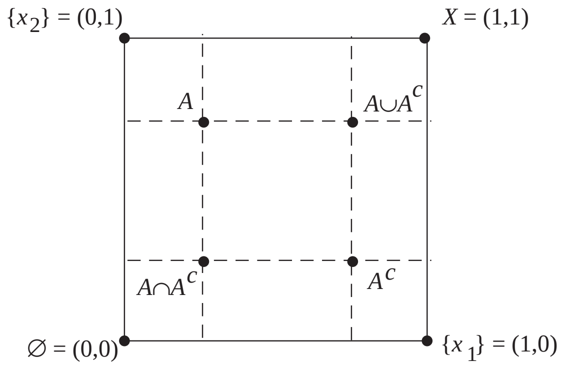


Fig. 11.5. Complement A^c of a fuzzy set

Kosko [259] establishes a direct relationship between the entropy of fuzzy sets and the geometric visualization of the union and intersection operations. To compute the entropy of a set, we need to determine the distance between the origin and the coordinates of the set. This distance is called the *cardinality* of the fuzzy set.

Definition 12. *Let A be a subset of a universal set X. The cardinality $|A|$ of A is the sum of the membership values of all elements of X with respect to A, i.e.,*

$$|A| = \sum_{x \in X} \mu_A(x)$$

This definition of cardinality corresponds to the distance of the representation of A from the origin using a *Manhattan metric*.

Figure 11.6 shows how to define the entropy of a set A using the cardinality of the sets $A \cap A^c$ and $A \cup A^c$. The Manhattan distances d_1 and d_2, introduced before to measure the fuzzy entropy, correspond to the cardinalities of the sets $A \cap A^c$ and $A \cup A^c$. The entropy concept introduced previously in an informal manner can then be formalized with our next definition.

Definition 13. *The real value*

$$E(A) = \frac{|A \cap A^c|}{|A \cup A^c|}$$

is called the entropy *of the fuzzy set A.*

The entropy of a crisp set is always zero, since for a crisp set $A \cap A^c = \emptyset$. In fuzzy set theory $E(A)$ is a value in the interval $[0, 1]$, since $A \cap A^c$ can be non-void.

Some authors take the geometric definition of entropy as given and derive Definition 13 as a theorem, which is called the *fuzzy entropy theorem.* [259]. Here we take the definition as given, since the geometric interpretation of fuzzy union and intersection depends on the exact definition of the fuzzy operators.

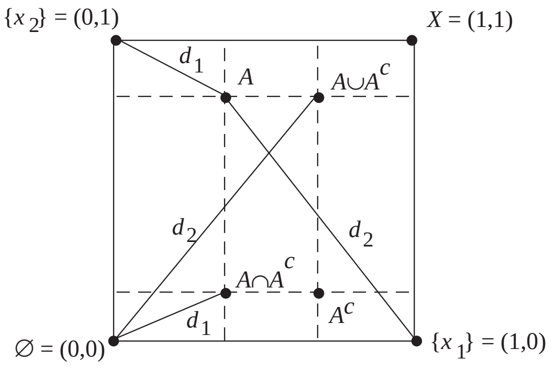


Fig. 11.6. Geometric representation of the entropy of a fuzzy set

11.1.4 Fuzzy set theory, logic operators, and geometry

It is known in mathematics that an isomorphism exists between set theory and classic propositional logic. In set theory, the three operators union, intersection, and complement $(\cup, \cap, c)$ allow the construction of new sets from other sets. In propositional logic, the operators OR, AND and NOT $(\vee, \wedge, \neg)$ are used to build new propositions.

The union operator of classical set theory can be constructed using the OR operator. Let A and B be two crisp sets, that is, $\mu_A, \mu_B : X \to \{0,1\}$. The membership function $\mu_{a \cup B}$ of the union set $A \cup B$ is

$$\mu_{A\cup B}(x) = \mu_A(x) \vee \mu_B(x) \quad \forall x \in X\,, \tag{11.4}$$

where the value 0 is interpreted as the logic value *false* and 1 as *true*. In a similar way it holds that for the intersection of the sets A and B

$$\mu_{A\cap B}(x) = \mu_A(x) \wedge \mu_B(x) \quad \forall x \in X. \tag{11.5}$$

For the complement A^c of the set A it holds that

$$\mu_{A^c}(x) = \neg\mu_A(x) \tag{11.6}$$

This correspondence between the operators of classical set theory and classical logic can be extended to the validity of some equations in both systems. The laws of de Morgan, for example, are valid in classical set theory as well as in classical logic:

$$(A \cup B)^c \equiv A^c \cap B^c \qquad \text{corresponds to} \qquad \neg(A \vee B) \equiv \neg A \wedge \neg B$$

$$(A \cap B)^c \equiv A^c \cup B^c \qquad \text{corresponds to} \qquad \neg(A \wedge B) \equiv \neg A \vee \neg B$$

A fuzzy logic can be also derived from fuzzy set theory by respecting the isomorphism mentioned above. The fuzzy AND, OR, and NOT operators must be defined in such a way that the same general kinds of relation exist between them and their equivalents in classical set theory and logic.

The straightforward approach is therefore to identify the OR operation ($\tilde{\vee}$) with the maximum function, AND ($\tilde{\wedge}$) with the minimum, and complementation ($\tilde{\neg}$) with the function $x \mapsto 1 - x$. Equations (11.1), (11.2), and (11.3) can be written as

$$\mu_{A \cup B}(x) = \mu_A(x) \,\tilde{\vee}\, \mu_B(x) \quad \forall x \in X \tag{11.7}$$

$$\mu_{A \cup B}(x) = \mu_A(x) \,\tilde{\vee}\, \mu_B(x) \quad \forall x \in X \tag{11.8}$$

$$\mu_{A^c}(x) = \tilde{\neg}\mu_A(x) \quad \forall x \in X \tag{11.9}$$

In this way an isomorphism between fuzzy set theory and fuzzy logic is constructed which preserves the properties of the isomorphism present in the classical theories.

Many rules of classical logic are still valid in the world of fuzzy operators. For example, the functions min and max are commutative and associative. However, the principle of no contradiction has been abolished. For a proposition A with truth value 0.4 we get

$$A \,\tilde{\wedge}\, \tilde{\neg} A = \min(0.4, 1 - 0.4) \neq 0$$

The principle of excluded middle is not valid for A either:

$$A \,\tilde{\vee}\, \tilde{\neg} A = \max(0.4, 1 - 0.4) \neq 1$$

11.1.5 Families of fuzzy operators

Up to this point we have worked with fuzzy operators defined in a rather informal way, since there are whole families of these operators that can be defined. Now we will give an axiomatic definition using the properties we would like the operators to exhibit.

Consider the fuzzy OR operator. In the fuzzy logic literature [248] such an operator is required to fulfill the following axioms:

- Axiom U1. Boundary conditions:

$$\begin{aligned} 0 \,\tilde{\vee}\, 0 &= 0 \\ 1 \,\tilde{\vee}\, 0 &= 1 \\ 0 \,\tilde{\vee}\, 1 &= 1 \\ 1 \,\tilde{\vee}\, 1 &= 1 \end{aligned}$$

- Axiom U2. Commutativity:

$$a \,\tilde{\vee}\, b = b \,\tilde{\vee}\, a$$

- Axiom U3. Monotonicity:

$$\text{If } a \leq a' \text{ and } b \leq b', \text{ then } a \mathbin{\tilde{\vee}} b \leq a' \mathbin{\tilde{\vee}} b'.$$

- Axiom U4. Associativity:

$$a \mathbin{\tilde{\vee}} (b \mathbin{\tilde{\vee}} c) = (a \mathbin{\tilde{\vee}} b) \mathbin{\tilde{\vee}} c$$

It is easy to show that the maximum function fulfills these four conditions. There are many other functions for which the four axioms are valid, for example

$$B(a, b) = \min(1, a + b)$$

which is called the *bounded sum* of a and b. An alternative fuzzy OR operator can be defined using this function.

However, the bounded sum is not idempotent, that is, in general $B(a, a) \neq a$. We can therefore introduce a fifth axiom to exclude such operators:

- Axiom U5. Idempotence:

$$a \mathbin{\tilde{\vee}} a = a$$

Depending on the axioms selected, different fuzzy operators and different logic systems can be defined. Consequently, the term *fuzzy logic* refers to a family of different theories and not to a unique system of logic.

In the case of the fuzzy operator $\tilde{\wedge}$ axioms are also formulated in such a way that fuzzy AND is monotonic, commutative, and associative. The boundary conditions are:

$$\begin{aligned} 0 \mathbin{\tilde{\wedge}} 0 &= 0 \\ 1 \mathbin{\tilde{\wedge}} 0 &= 0 \\ 0 \mathbin{\tilde{\wedge}} 1 &= 0 \\ 1 \mathbin{\tilde{\wedge}} 1 &= 1 \end{aligned}$$

Idempotence can be demanded and can be enforced using a fifth axiom. For the fuzzy negation we use the following three axioms:

- Axiom N1. Boundary conditions:

$$\tilde{\neg} 1 = 0$$

$$\tilde{\neg} 0 = 1$$

- Axiom N2. Monotonicity:

$$\text{If } a \leq b \text{ then } \tilde{\neg} b \leq \tilde{\neg} a.$$

- Axiom N3. Involution:

$$\tilde{\neg}\tilde{\neg} a = a$$

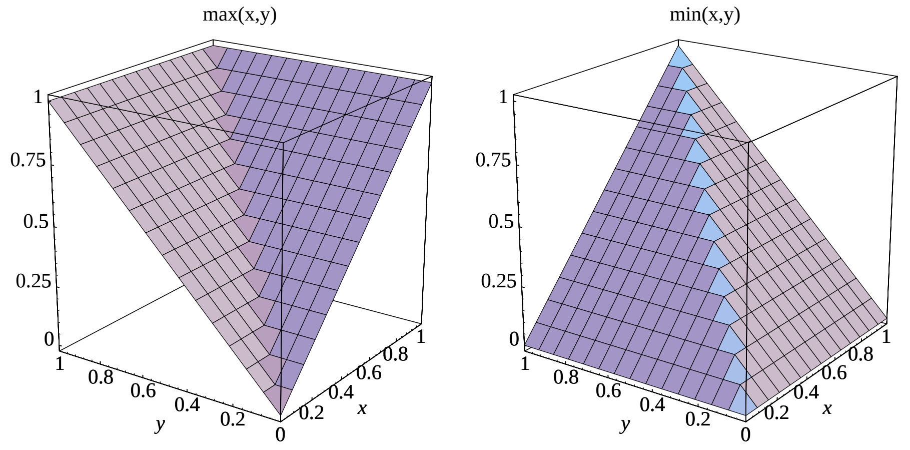


Fig. 11.7. The max and min functions

The difference between these fuzzy operators can be better visualized by looking at their graphs. Figure 11.7 shows the graphs of the functions max and min, that is, a possible fuzzy AND and fuzzy OR combination. They could also be used as activation functions in neural networks. Using output functions derived from fuzzy logic can have the added benefit of providing a logical interpretation of the neural output.

The graphs of the functions *bounded sum* and *bounded difference* are shown in Figure 11.8. Both fulfill the conditions imposed by the first four axioms for fuzzy OR and fuzzy AND operators, but are not idempotent.

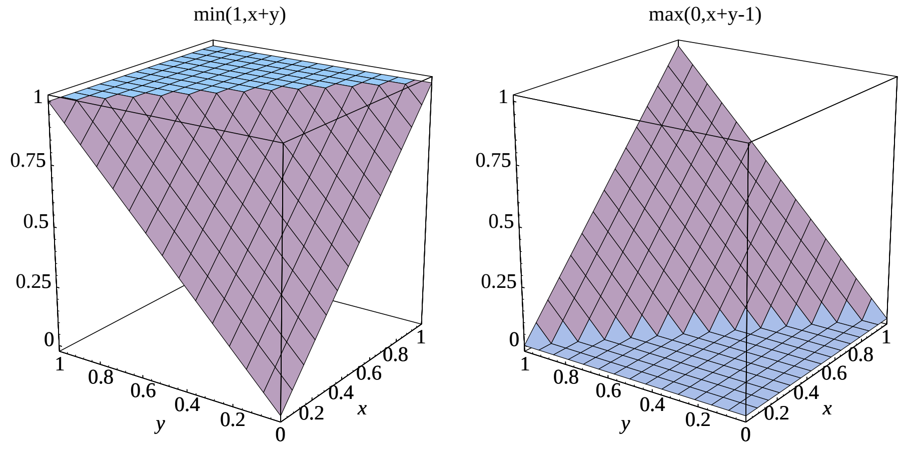


Fig. 11.8. The fuzzy operators *bounded sum* and *bounded difference*

It is also possible to define a parameterized family of fuzzy operators. Figure 11.9 illustrates this approach for the case of the so-called Yager union function which is given by

$$Y_p(a,b) = \min(1, (a^p + b^p)^{1/p}) \quad \text{for} \quad p \geq 1,$$

where a and b are real numbers in the interval $[0,1]$. The formula describes a family of operators. For $p = 2$, the function is an approximation of the bounded sum operator. For $p \gg 1$, Y_p is an approximation of the max function. Adjusting the parameter p we can select the desired variant of fuzzy logic.

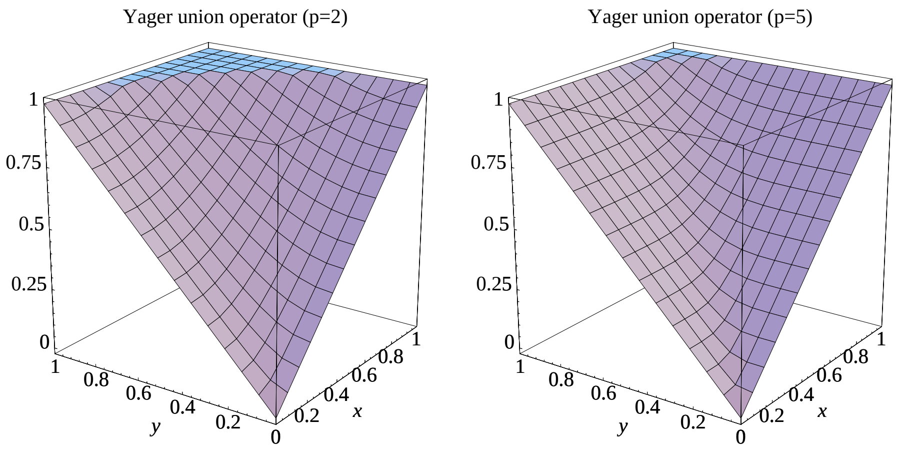


Fig. 11.9. Two variants of the Yager union operator

Figure 11.9 shows that the Yager union operator is not idempotent. If it were, the diagonal from the point $(0,0)$ to the point $(1,1)$ would belong to the graph of the function. This is the case for the functions min and max. It can be shown that these functions are the only ones in which the five axioms for fuzzy OR and fuzzy AND fully apply [248].

The geometric properties of fuzzy operators can be derived from the axioms for fuzzy OR, fuzzy AND, and fuzzy negation operators. The boundary conditions determine four values of the function. The commutativity of the operators forces the graph of the functions to be symmetrical with respect to the plane normal to the xy plane and which cuts it at the 45-degree diagonal. Monotonicity of the operators allows only those function graphs that do not fold. Associativity is more difficult to visualize but it roughly indicates that the function does not grow abruptly in some regions and stagnate in others. If all or some of these symmetry properties hold for a binary function, then this function fulfills the operator axioms and can be used as a fuzzy operator. The symmetry properties, in turn, lead to useful algebraic properties of the operators, and the connection between set theory, logic, and geometry is readily perceived.

11.2 Fuzzy inferences

Fuzzy logic operators can be used as the basis for inference systems. Such fuzzy inference methods have been extensively studied by the expert systems community. Knowledge that can only be formulated in a fuzzy, imprecise manner can be captured in rules that can be processed by a computer.

11.2.1 Inferences from imprecise data

Fuzzy inference rules have the same structure as classical ones. The rules R_1 and R_2, for example, may have the form

$$\begin{array}{l} R_1\text{:If}(A \mathbin{\tilde{\wedge}} B)\text{then } C. \\ R_2\text{:If}(A \mathbin{\tilde{\vee}} B)\text{then } D. \end{array}$$

The difference in conventional inference rules is the semantics of the fuzzy operators. In this section we identify the fuzzy operators $\tilde{\wedge}$ and $\tilde{\vee}$ with the functions min and max respectively.

Let the truth values of A and B be 0.4 and 0.7 respectively. In this case

$$A \mathbin{\tilde{\wedge}} B = \min(0.4, 0.7) = 0.4$$

$$A \mathbin{\tilde{\vee}} B = \max(0.4, 0.7) = 0.7$$

This is interpreted by the fuzzy inference mechanism as meaning that the rules R_1 and R_2 can only be partially applied, that is rule R_1 is applied to 40% and rule R_2 to 70%. The result of the inference is a combination of the propositions C and D.

Let us consider another example. Assume that a temperature controller must regulate an electrical heater. We can formulate three rules for such a system:

$$\begin{array}{lcl} R_1\text{:If} & (\text{temperature} = \text{cold}) & \text{then heat.} \\ R_2\text{:If} & (\text{temperature} = \text{normal}) & \text{then maintain.} \\ R_3\text{:If} & (\text{temperature=warm}) & \text{then reduce power.} \end{array}$$

Assume that a temperature of 12 degrees Celsius has a membership degree of 0.5 in relation to the set of cold temperatures and a membership degree of 0.3 in relation to the temperatures classified as normal. The temperature of 12 degrees is converted first of all into a fuzzy category which is the list of membership values of an element x of X in relation to previously selected fuzzy sets of the universal set X.

The fuzzy category T can be expressed using a similar notation as for fuzzy sets. In our example:

$$T = \mathit{cold}/0.5 + \mathit{normal}/0.3 + \mathit{warm}/0.0.$$

Note the difference in the notation for fuzzy sets. If a fuzzy category x is defined in relation to fuzzy sets A, B, and C, it is written as

$$x = A/\mu_A(x) + B/\mu_B(x) + C/\mu_C(x)$$

and not as $x = \mu_A(x)/A + \mu_B(x)/B + \mu_C(x)/C$.

Using T we can now evaluate the rules $R1$, $R2$, and $R3$ in parallel. The result is that each rule is valid to a certain extent. A *fuzzy inference* is the combination of the three possible consequences, weighted according to their validity. The result of a fuzzy inference is therefore a fuzzy category. In our example we deduce that

$$\mathit{action} = \mathit{heat}/0.5 + \mathit{maintain}/0.3 + \mathit{reduce}/0.0.$$

Fuzzy inference systems compute inferences of this type. Imprecise data, which is represented by fuzzy categories, leads to new fuzzy categories which represent the conclusion. In expert systems this kind of result can be processed further or it can be transformed into a crisp value. In the case of an electronic fuzzy controller this last step is always necessary.

The advantage of formulating fuzzy inference rules is their low granularity. In many cases apparently complex control problems can be modeled using just a few rules. If we tried to express all actions as consequences of exact numerical rules, we would have to write more of them or make each one much more complex.

11.2.2 Fuzzy numbers and inverse operation

The example in the last section shows that a fuzzy controller operates, in general, in three steps: a) A measurement is transformed into a fuzzy category using the membership functions of all defined categories; b) All pertinent inference rules of the control system are evaluated and a fuzzy inference is produced; c) In the last step the result of the fuzzy inference is transformed into a crisp value.

There are several alternative ways to transform a measurement into fuzzy categories. A frequent approach is to use triangular or trapezium-shaped membership functions. Figure 11.10 shows how a measurement interval can be subdivided using triangular-shaped membership functions and Figure 11.11 shows the same kind of subdivision but with trapezium-shaped membership functions.

The transformation of a measurement x into a fuzzy category is given by the membership values $\alpha_1, \alpha_2, \alpha_3$ derived from the membership functions (as shown in Figure 11.10).

An important problem is how to transform the membership values $\alpha_1, \alpha_2, \alpha_3$ back into the measurement x, that is, how to implement the inverse operation to the fuzzifying of the crisp number. A popular approach is the centroid method. Figure 11.12 shows the value x and its transformation into $\alpha_1, \alpha_2, \alpha_3$. From these three values we can reconstruct the original x. To do this, the surfaces of the triangular regions limited by the heights α_1, α_2

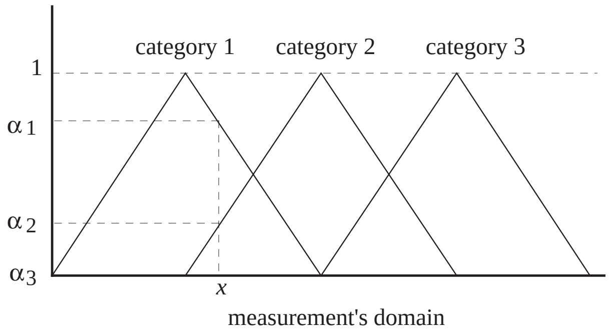


Fig. 11.10. Categories with triangular membership functions

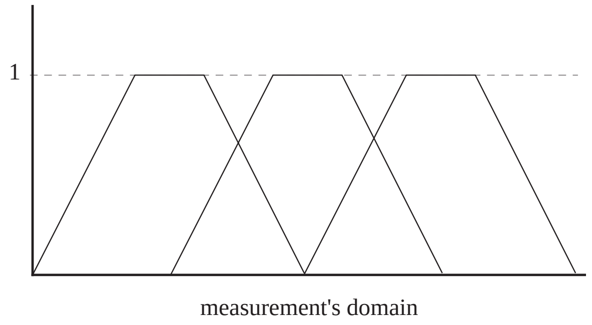


Fig. 11.11. Categories with trapezium-shaped membership functions

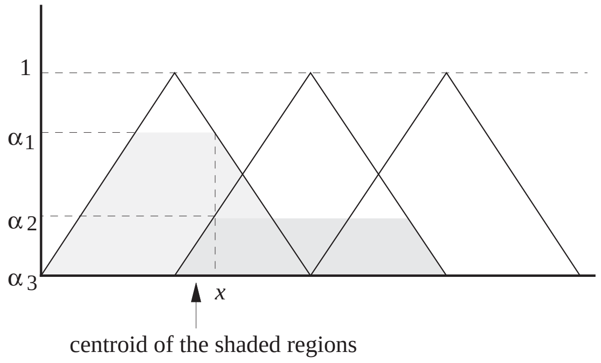


Fig. 11.12. The centroid method

and α_3 are computed. The *horizontal component of the centroid* of the total surface is the approximation to x we are looking for (Figure 11.12).

For all x values for which at least two of the three numbers $\alpha_1, \alpha_2, \alpha_3$ are different from zero, we can compute a good approximation using the centroid method. Figure 11.13 shows the difference between x and its approximation when the basis of the triangular regions is of length 2, their height is 1 and the arrangement is the one shown in Figure 11.12. The value of x has been chosen in the interval $[1, 2]$. Figure 11.13 shows that the relative difference from the correct value of x is never greater than 10%.

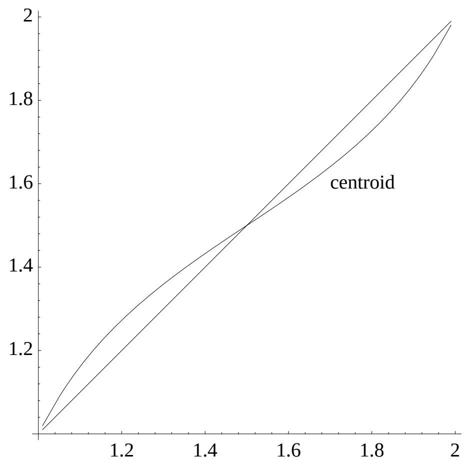


Fig. 11.13. Reconstruction error of the centroid method

The centroid method produces better or worse inverse transformations depending on the placement of the triangular categories. Weighting the surfaces of the triangular regions according to their position can also affect how good the inverse transformation is.

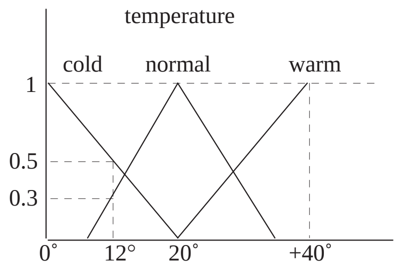
11.3 Control with fuzzy logic

A fuzzy controller is a regulating system whose modus operandi is specified with fuzzy rules. In general it uses a small set of rules. The measurements are processed in their fuzzified form, fuzzy inferences are computed, and the result is defuzzified, that is, it is transformed back into a specific number.

11.3.1 Fuzzy controllers

The example with the electrical heater will be completed in this section. We must determine the domain of definition of the variables used in the problem. Assume that the room temperature is a number between 0 and 40 degrees Celsius. The controller can vary the electrical power consumed between 0 and 100 (in some suitable units), whereby 50 is the normal stand-by value.

Figure 11.14 shows the membership functions for the temperature categories “cold”, “normal”, and “warm” and the control categories “reduce”, “maintain”, and “heat”.



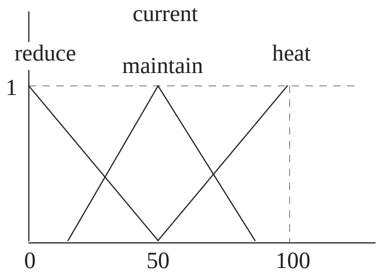


Fig. 11.14. Membership functions for temperature and electric current categories

The temperature of 12 degrees corresponds to the fuzzy number $T = \mathit{cold}/0.5 + \mathit{normal}/0.3 + \mathit{warm}/0.0$. These values lead to the previously computed inference $\mathit{action} = \mathit{heat}/0.5 + \mathit{maintain}/0.3 + \mathit{reduce}/0.0$. The controller must transform the result of this fuzzy inference into a definite value. The surfaces of the membership triangles below the inferred degree of membership are calculated. The light shaded surface in Figure 11.15 corresponds to the action "heat", which is valid to 50%. The darker region corresponds to the action "maintain" that is valid to 30%. The centroid of the two shaded regions lies at about 70. This value for the power consumption is adjusted by the controller in order to heat the room.

It is of course possible to formulate more complex rules involving more than two variables. In all cases, though, we have to evaluate all rules simultaneously. Kosko shows some examples of dynamical systems with three or more control variables [259].

11.3.2 Fuzzy networks

Fuzzy systems can be represented as networks. The computing units must implement fuzzy operators. Figure 11.16 shows a network with four hidden units. Each one of them receives the inputs x_1, x_2 and x_3 which correspond to the fuzzy categorization of a specific number. The fuzzy operators are evaluated in parallel in the hidden layer of the network, which corresponds to the set of inference rules. The last unit in the network is the defuzzifier,

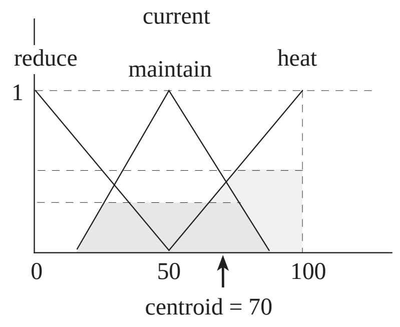


Fig. 11.15. Centroid computation

which transforms the fuzzy inferences into a specific control variable. The importance of each fuzzy inference rule is weighted by the numbers α_1, α_2, and α_3 as in a weighted centroid computation.

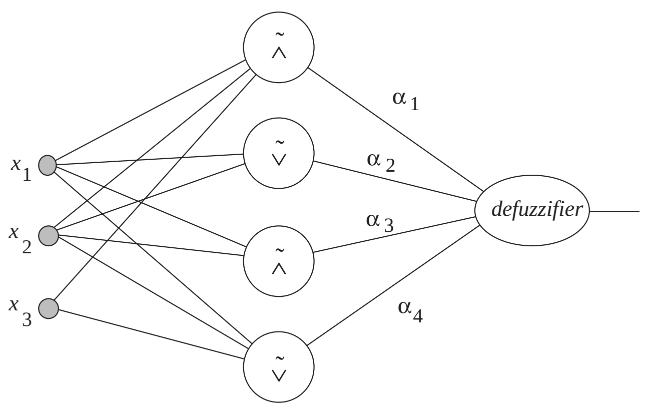


Fig. 11.16. Example of a fuzzy network

More complex rules can be implemented and this can lead to networks with several layers. However, fuzzy systems do not usually lead to very deep networks. Since at each fuzzy inference step the precision of the conclusion is reduced, it is not advisable to build too long an inference chain.

Fuzzy operators cannot be computed exactly by sigmoidal units, but for some of them a relatively good approximation is possible, for example, for bounded sum or bounded difference. A fuzzy inference chain using these operators can be approximated by a neural network.

The defuzzifier operator in the last layer can be approximated with standard units. If the membership functions are triangles, the surface of the triangles grows quadratically with the height. A quadratic function of this form can be approximated in the pertinent interval using sigmoids. The parameters of the approximation can be set with the help of a learning algorithm.

11.3.3 Function approximation with fuzzy methods

A fuzzy controller is just a system for the rapid computation of an approximation of a coarsely defined control surface, like the one shown in Figure 11.17. The fuzzy controller computes a control variable according to the values of the variables x and y. Both variables are transformed into fuzzy categories. Assume that each variable is transformed into a combination of three categories. There are nine different combinations of the categories for x and y. For each of these nine combinations the value of the control variable is defined. This fixes nine points of the control surface.

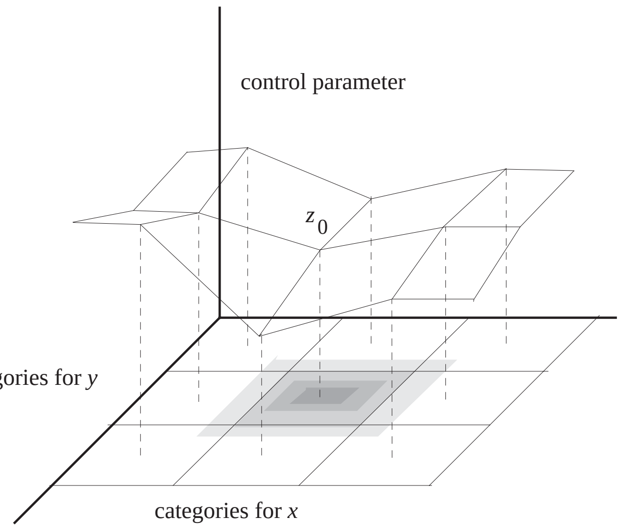


Fig. 11.17. Approximation of a control surface

Arbitrary values of x and y belong, to different degrees, to the nine combined categories. This means that for arbitrary combinations of x and y an interpolation of the known function values of the control variable is needed. A fuzzy controller performs this computation according to the degree of membership of (x, y) in each combined category. In Figure 11.17 the different shadings of the quadratic regions in the xy plane represent the membership of the input in the category for which the control variable assumes the value z_0. Other values, which correspond to the lighter shaded regions receive a value for the control variable which is an interpolation of the neighboring z-values.

The control surface can be defined using a few points and, if the control function is smooth, a good approximation to other values is obtained with simple interpolation. The reduced number of given points corresponds to a reduced number of inference rules in the fuzzy controller. The advantage of such an approach lies in the *economic use of rules*. Inference rules can be defined in the fuzzy formalism in a straightforward manner. The interpolation mechanism is taken as given. This approach works of course only in the case where the control function has an adequate degree of smoothness.

11.3.4 The eye as a fuzzy system – color vision

It is interesting to note that the eye makes use of a similar strategy to fuzzy controllers with regard to color vision. Photons of different wavelengths, corresponding to different colors in the visible spectrum, impinge on the retina. The eye contains receptors for only three types of colors. We can find in the cochlea different receptors for many of the frequencies present in sounds that we can hear, but in the eyes we find only receptors that are maximally excited with light from the spectral regions corresponding to the colors blue, green, and red. That is why the receptors have received the name of the colors they detect. Color vision must accommodate sensors to process a two-dimensional image at every pixel and this can only be done by reducing the number of detector types available.

The visible spectrum for humans extends from 400 up to 650 nanometers wavelength. A monochromatic color, that is, a color with a definite and crisp wavelength, excites all three receptor types in the retina. The output of each receptor class, however, is not identical but depends on the wavelength of the light. It has been shown in many color vision experiments, and later through direct measurements, that the ouput functions of the three receptor classes correspond to those shown in Figure 11.18. Blue receptors, for example, reach their maximal excitation for wavelengths around 430 nm. Green receptors respond maximally at 530 nm and red receptors at 560 nm. When monochromatic light excites the receptors on the retina its wavelength is transformed into three different excitation levels, that is, into a relative excitation of the three receptor types. The wavelength is transformed into a fuzzy category, just as in the case of fuzzy controllers. The three excitation levels measure the degree of membership in each of the three color categories blue, green, and red. The subsequent processing of the color information is performed based on this and additional coding steps (for example, comparing complementary colors). This is why a mixture of two colors is perceived by the brain as a third color.

Some simple physiological considerations show that good color discrimination requires at least three types of receptors [205]. Coding of the wavelength using three excitation values reduces the number of rules needed in subsequent processing. The sparseness of rules in fuzzy controllers finds its equivalent here in the sparseness of the biological organs.

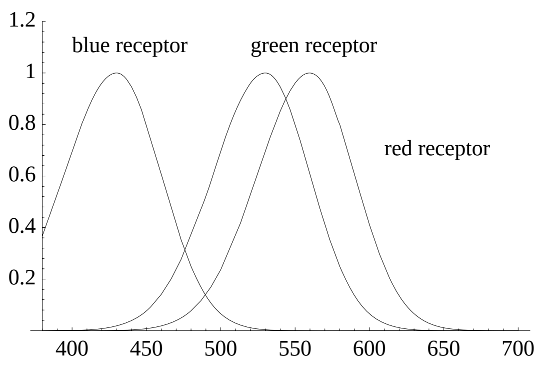


Fig. 11.18. Response function of the three receptors in the retina

11.4 Historical and bibliographical remarks

Multiple-valued logic has a long history [362]. Aristotle raised the question of whether all valid propositions can only be assigned the logical values true or false. The first attempts at formulating a multiple-valued logic were made by logicians such as Charles Sanders Pierce at the end of the nineteenth and beginning of the twentieth century. The first well-known system of multiple valued logic was introduced by the Pole Jan Lukasiewicz in the 1920s. By defining a third truth value Lukasiewicz created a system of logic which was later axiomatized by other authors [362]. From 1930 onwards, renowned mathematicians such as Gödel, Brouwer, and von Neumann continued work on developing an alternative system of logic which could be used in mathematics or physics. In their investigations they considered the possibility of an infinite number of truth values.

Fuzzy logic, as formulated by Zadeh in 1965, is a multiple-valued logic with a continuum of truth values. The term fuzzy logic really refers more to a whole family of possible logic theories which vary in the definition of their logical operators [465]. In the 1970s the interest in fuzzy logic and its possible use in expert systems increased, so that the number of papers published on this topic increased almost exponentially from year to year [Gaines 1977]. First attempts to use fuzzy logic for control systems were extensively examined by Mamdani's group in England in the 1970s [Mamdani 1977]. Since then fuzzy controllers have left the research laboratories and are used in industrial and consumer electronics.

Over the last few years interest in fuzzy controllers has increased dramatically. Some companies already offer microchips with hardwired fuzzy operators and fuzzy inference rules. It is estimated that worldwide sales of fuzzy chips will increase from 1.5 billions dollars in 1990 to 13 billion dollars in the year 2000. Some companies are planning to incorporate fuzzy operators in the instruction set of normal microprocessors.

Fuzzy logic offers an interpretation model for the analysis of neural networks. Some pioneer work in this field has been carried out by Bart Kosko. Over the last few years other authors have continued with the examination of the relationships between fuzzy logic and neural networks [273]. An active field of research is the formulation of learning algorithms for fuzzy systems which retain the clarity of the fuzzy formalism.

Exercises

1. Show that the maximum function fulfills the axioms U1-U5.
2. What are the corresponding axioms for the fuzzy intersection? Show that the minimum function fulfills them.
3. Propose a learning algorithm for a fuzzy network like the one shown in Figure 11.16.
4. Construct a set of fuzzy control rules for the pole balancing car shown in Figure 15.18.
5. Are triangular-shaped membership functions better or worse than trapezium-shaped functions? Assume that a crisp number is transformed into fuzzy categories and then retransformed into a crisp number using the centroid method. What kind of function produces the lowest reconstruction error?

12

Associative Networks

12.1 Associative pattern recognition

The previous chapters were devoted to the analysis of neural networks without feedback, capable of mapping an input space into an output space using only feed-forward computations. In the case of backpropagation networks we demanded continuity from the activation functions at the nodes. The neighborhood of a vector $\mathbf{x}$ in input space is therefore mapped to a neighborhood of the image $\mathbf{y}$ of $\mathbf{x}$ in output space. It is this property which gives its name to the *continuous mapping networks* we have considered up to this point.

In this chapter we deal with another class of systems, known generically as associative memories. The goal of learning is to *associate* known input vectors with given output vectors. Contrary to continuous mappings, the neighborhood of a known input vector $\mathbf{x}$ should also be mapped to the image $\mathbf{y}$ of $\mathbf{x}$, that is if $B(\mathbf{x})$ denotes all vectors whose distance from $\mathbf{x}$ (using a suitable metric) is lower than some positive constant ε, then we expect the network to map $B(\mathbf{x})$ to $\mathbf{y}$. Noisy input vectors can then be associated with the correct output.

12.1.1 Recurrent networks and types of associative memories

Associative memories can be implemented using networks with or without feedback, but the latter produce better results. In this chapter we deal with the simplest kind of feedback: the output of a network is used repetitively as a new input until the process converges. However, as we will see in the next chapter not all networks converge to a stable state after having been set in motion. Some restrictions on the network architecture are needed.

The function of an associative memory is to recognize previously learned input vectors, even in the case where some noise has been added. We have already discussed a similar problem when dealing with clusters of input vectors (Chap. 5). The approach we used there was to find cluster centroids in input space. For an input $\mathbf{x}$ the weight vectors produced a maximal excitation at the unit representing the cluster assigned to $\mathbf{x}$, whereas all other units remained silent. A simple approach to inhibit the other units is to use non-local information to decide which unit should fire.

The advantage of associative memories compared to this approach is that only the local information stream must be considered. The response of each unit is determined exclusively by the information flowing through its own weights. If we take the biological analogy seriously or if we want to implement these systems in VLSI, locality is always an important goal. And as we will see in this chapter, a learning algorithm derived from biological neurons can be used to train associative networks: it is called *Hebbian learning.*

The associative networks we want to consider should not be confused with conventional associative memory of the kind used in digital computers, which consists of content addressable memory chips. With this in mind we can distinguish between three overlapping kinds of associative networks [294, 255]:

- *Heteroassociative networks* map m input vectors $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^m$ in n-dimensional space to m output vectors $\mathbf{y}^1, \mathbf{y}^2, \ldots, \mathbf{y}^m$ in k-dimensional space, so that $\mathbf{x}^i \mapsto \mathbf{y}^i$. If $\|\tilde{\mathbf{x}} - \mathbf{x}^i\|^2 < \varepsilon$ then $\tilde{\mathbf{x}} \mapsto \mathbf{y}^i$. This should be achieved by the learning algorithm, but becomes very hard when the number m of vectors to be learned is too high.
- *Autoassociative networks* are a special subset of the heteroassociative networks, in which each vector is associated with itself, i.e., $\mathbf{y}^i = \mathbf{x}^i$ for $i = 1, \ldots, m$. The function of such networks is to correct noisy input vectors.
- *Pattern recognition networks* are also a special type of heteroassociative networks. Each vector $\mathbf{x}^i$ is associated with the scalar i. The goal of such a network is to identify the 'name' of the input pattern.

Figure 12.1 shows the three kinds of networks and their intended use. They can be understood as automata which produce the desired output whenever a given input is fed into the network.

12.1.2 Structure of an associative memory

The three kinds of associative networks discussed above can be implemented with a single layer of computing units. Figure 12.2 shows the structure of a heteroassociative network without feedback. Let w_{ij} be the weight between input site i and unit j. Let $\mathbf{W}$ denote the $n \times k$ weight matrix $[w_{ij}]$. The row vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ produces the excitation vector $\mathbf{e}$ through the computation

$$\mathbf{e} = \mathbf{x}\mathbf{W}.$$

The activation function is computed next for each unit. If it is the identity, the units are just linear associators and the output $\mathbf{y}$ is just $\mathbf{x}\mathbf{W}$. In general m different n-dimensional row vectors $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^m$ have to be associated

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

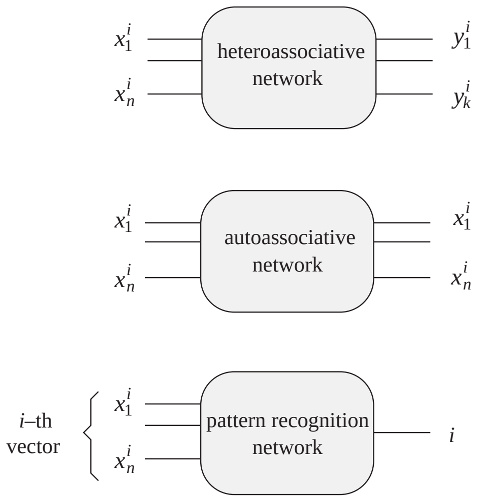


Fig. 12.1. Types of associative networks

with m k-dimensional row vectors $\mathbf{y}^1, \mathbf{y}^2, \ldots, \mathbf{y}^m$. Let $\mathbf{X}$ be the $m \times n$ matrix whose rows are each one of the input vectors and let $\mathbf{Y}$ be the $m \times k$ matrix whose rows are the output vectors. We are looking for a weight matrix $\mathbf{W}$ for which

$$\mathbf{XW} = \mathbf{Y} \tag{12.1}$$

holds. In the autoassociative case we associate each vector with itself and equation (12.1) becomes

$$\mathbf{XW} = \mathbf{X} \tag{12.2}$$

If $m = n$, then $\mathbf{X}$ is a square matrix. If it is invertible, the solution of (12.1) is

$$\mathbf{W} = \mathbf{X}^{-1}\mathbf{Y},$$

which means that finding $\mathbf{W}$ amounts to solving a linear system of equations.

What happens now if the output of the network is used as the new input? Figure 12.3 shows such a recurrent autoassociative network. We make the assumption that all units compute their outputs simultaneously and we call such networks *asynchronous*. In each step the network is fed an input vector $\mathbf{x}(i)$ and produces a new output $\mathbf{x}(i+1)$. The question with this class of networks is whether there is a fixed point $\vec{\xi}$ such that

$$\vec{\xi}\,\mathbf{W} = \vec{\xi}\ .$$

The vector $\vec{\xi}$ is an eigenvector of $\mathbf{W}$ with eigenvalue 1. The network behaves as a first-order dynamical system, since each new state $\mathbf{x}(i+1)$ is completely determined by its most recent predecessor [30].

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

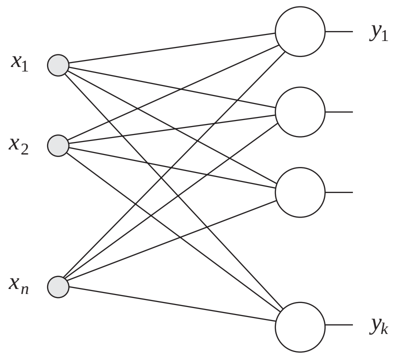


Fig. 12.2. Heteroassociative network without feedback

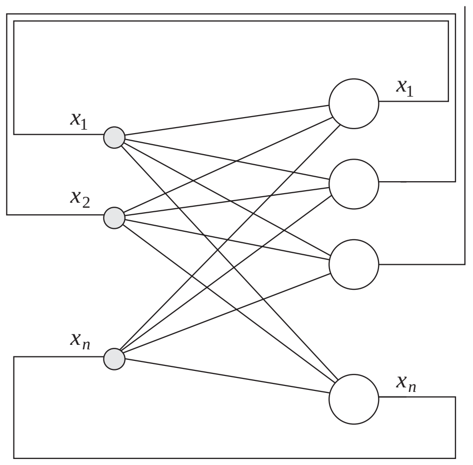


Fig. 12.3. Autoassociative network with feedback

12.1.3 The eigenvector automaton

Let **W** be the weight matrix of an autoassociative network, as shown in Figure 12.3. The individual units are linear associators. As we said before, we are interested in fixed points of the dynamical system but not all weight matrices lead to convergence to a stable state. A simple example is a rotation by 90 degrees in two-dimensional space, given by the matrix

$$\mathbf{W} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}.$$

Such a matrix has no non-trivial eigenvectors. There is no fixed point for the dynamical system, but infinitely many cycles of length four. By changing the angle of rotation, arbitrarily long cycles can be produced and by picking an irrational angle, even a non-cyclic succession of states can be produced.

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

Quadratic matrices with a complete set of eigenvectors are more useful for storage purposes. An $n \times n$ matrix $\mathbf{W}$ has at most n linear independent eigenvectors and n eigenvalues. The eigenvectors $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^n$ satisfy the set of equations

$$\mathbf{x}^i \mathbf{W} = \lambda_i \mathbf{x}^i \qquad \text{for } i = 1, \ldots, n,$$

where $\lambda_1, \ldots, \lambda_n$ are the matrix eigenvalues.

Each weight matrix with a full set of eigenvectors defines a kind of "eigenvector automaton". Given an initial vector, the eigenvector with the largest eigenvalue can be found (if it exists). Assume without loss of generality that λ_1 is the eigenvalue of $\mathbf{w}$ with the largest magnitude, that is, $|\lambda_1| > |\lambda_i|$ for $i \neq j$. Let $\lambda_1 > 0$ and pick randomly a non-zero n-dimensional vector $\mathbf{a}_0$. This vector can be expressed as a linear combination of the n eigenvectors of the matrix $\mathbf{W}$:

$$\mathbf{a}_0 = \alpha_1 \mathbf{x}^1 + \alpha_2 \mathbf{x}^2 + \cdots + \alpha_n \mathbf{x}^n.$$

Assume that all constants α are non-zero. After the first iteration with the weight matrix $\mathbf{W}$ we get

$$\begin{aligned}\mathbf{a}_1 &= \mathbf{a}_0 \mathbf{W} \\ &= (\alpha_1 \mathbf{x}^1 + \alpha_2 \mathbf{x}^2 + \cdots + \alpha_n \mathbf{x}^n) \mathbf{W} \\ &= \alpha_1 \lambda_1 \mathbf{x}^1 + \alpha_2 \lambda_2 \mathbf{x}^2 + \cdots + \alpha_n \lambda_n \mathbf{x}^n.\end{aligned}$$

After t iterations the result is

$$\mathbf{a}_t = \alpha_1 \lambda_1^t \mathbf{x}^1 + \alpha_2 \lambda_2^t \mathbf{x}^2 + \cdots + \alpha_n \lambda_n^t \mathbf{x}^n.$$

It is obvious that the eigenvalue λ_1, the one with the largest magnitude, dominates this expression for a big enough t. The vector $\mathbf{a}_t$ can be brought arbitrarily close to the eigenvector $\mathbf{x}^1$ (with respect to the direction and without considering the relative lengths). In each iteration of the associative network, the vector $\mathbf{x}^1$ attracts any other vector $\mathbf{a}_0$ whose component α_1 is non-zero. An example is the following weight matrix:

$$\mathbf{W} = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$$

Two eigenvectors of this matrix are $(1, 0)$ and $(0, 1)$ with respective eigenvalues 2 and 1. After t iterations any initial vector (x_1, x_2) with $x_1 \neq 0$ is transformed into the vector $(2^t x_1, x_2)$, which comes arbitrarily near to the vector $(1, 0)$ for a large enough t. In the language of the theory of dynamical systems, the vector $(1, 0)$ is an *attractor* for all those two-dimensional vectors whose first component does not vanish.

12.2 Associative learning

The simple examples of the last section illustrate our goal: we want to use associative networks as dynamical systems, whose attractors are exactly those

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

vectors we would like to store. In the case of the linear eigenvector automaton, unfortunately just one vector absorbs almost the whole of input space. The secret of associative network design is locating as many attractors as possible in input space, each one of them with a well-defined and bounded influence region. To do this we must introduce a nonlinearity in the activation of the network units so that the dynamical system becomes nonlinear.

Bipolar coding has some advantages over binary coding, as we have already seen several times, but this is still more noticeable in the case of associative networks. We will use a step function as nonlinearity, computing the output of each unit with the sign function

$$\operatorname{sgn}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

The main advantage of bipolar coding is that bipolar vectors have a greater probability of being orthogonal than binary vectors. This will be an issue when we start storing vectors in the associative network.

12.2.1 Hebbian learning – the correlation matrix

Consider a single-layer network of k units with the sign function as activation function. We want to find the appropriate weights to map the n-dimensional input vector $\mathbf{x}$ to the k-dimensional output vector $\mathbf{y}$. The simplest learning rule that can be used in this case is Hebbian learning, proposed in 1949 by the psychologist Donald Hebb [182]. His idea was that two neurons which are simultaneously active should develop a degree of interaction higher than those neurons whose activities are uncorrelated. In the latter case the interaction between the elements should be very low or zero [308].

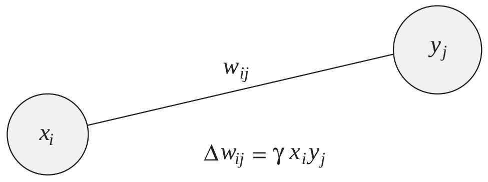


Fig. 12.4. The Hebb rule

In the case of an associative network, Hebbian learning is applied, updating the weight w_{ij} by Δw_{ij}. This increments measures the correlation between the input x_i at site i and the output y_j of unit j. During learning, both input and output are clamped to the network and the weight update is given by

$$\Delta w_{ij} = \gamma\, x_i x_j .$$

The factor γ is a learning constant in this equation. The weight matrix $\mathbf{W}$ is set to zero before Hebbian learning is started. The learning rule is applied

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

to all weights clamping the n-dimensional row vector $\mathbf{x}^1$ to the input and the k-dimensional row vector $\mathbf{y}^1$ to the output. The updated weight matrix $\mathbf{W}$ is the correlation matrix of the two vectors and has the form

$$\mathbf{W} = [w_{ij}]_{n\times k} = \left[x_i^1 y_j^1\right]_{n\times k}.$$

The matrix $\mathbf{W}$ maps the non-zero vector $\mathbf{x}^1$ exactly to the vector $\mathbf{y}^1$, as can be proved by looking at the excitation of the k output units of the network, which is given by the components of

$$\begin{aligned}\mathbf{x}^1\mathbf{W} &= (y_1^1 \sum_{i=1}^n x_i^1 x_i^1, y_2^1 \sum_{i=1}^n x_i^1 x_i^1, ..., y_k^1 \sum_{i=1}^n x_i^1 x_i^1) \\ &= \mathbf{y}^1(\mathbf{x}^1 \cdot \mathbf{x}^1).\end{aligned}$$

For $\mathbf{x}^1 \neq \mathbf{0}$ it holds $\mathbf{x}^1 \cdot \mathbf{x}^1 > 0$ and the output of the network is

$$\operatorname{sgn}(\mathbf{x}^1\mathbf{W}) = (y_1^1, y_2^1, ..., y_k^1) = \mathbf{y}^1,$$

where the sign function is applied to each component of the vector of excitations.

In general, if we want to associate m n-dimensional non-zero vectors $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^m$ with m k-dimensional vectors $\mathbf{y}^1, \mathbf{y}^2, \ldots, \mathbf{y}^m$, we apply Hebbian learning to each input-output pair and the resulting weight matrix is

$$\mathbf{W} = \mathbf{W}^1 + \mathbf{W}^2 + \cdots + \mathbf{W}^m, \tag{12.3}$$

where each matrix $\mathbf{W}^\ell$ is the $n \times k$ correlation matrix of the vectors $\mathbf{x}^\ell$ and $\mathbf{y}^\ell$, i.e.,

$$\mathbf{W}^\ell = \left[x_i^\ell y_j^\ell\right]_{n\times k}.$$

If the input to the network is the vector $\mathbf{x}^p$, the vector of unit excitations is

$$\begin{aligned}\mathbf{x}^p\mathbf{W} &= \mathbf{x}^p(\mathbf{W}^1 + \mathbf{W}^2 + \cdots + \mathbf{W}^m) \\ &= \mathbf{x}^p\mathbf{W}^p + \sum_{\ell\neq p}^m \mathbf{x}^p\mathbf{W}^\ell \\ &= \mathbf{y}^p(\mathbf{x}^p \cdot \mathbf{x}^p) + \sum_{\ell\neq p}^m \mathbf{y}^\ell(\mathbf{x}^\ell \cdot \mathbf{x}^p).\end{aligned}$$

The excitation vector is equal to $\mathbf{y}^p$ multiplied by a positive constant plus an additional perturbation term $\sum_{\ell\neq p}^m \mathbf{y}^\ell(\mathbf{x}^\ell \cdot \mathbf{x}^p)$ called the *crosstalk*. The network produces the desired vector $\mathbf{y}^p$ as output when the crosstalk is zero. This is the case whenever the input patterns $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^m$ are pairwise orthogonal. Yet, the perturbation term can be different from zero and the mapping can still work. The crosstalk should only be sufficiently smaller than $\mathbf{y}^p(\mathbf{x}^p \cdot \mathbf{x}^p)$. In that case the output of the network is

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

$$\mathrm{sgn}(\mathbf{x}^p\mathbf{W}) = \mathrm{sgn}\left(\mathbf{y}^p(\mathbf{x}^p \cdot \mathbf{x}^p) + \sum_{\ell \neq p}^{m} \mathbf{y}^\ell(\mathbf{x}^\ell \cdot \mathbf{x}^p)\right).$$

Since $\mathbf{x}^p \cdot \mathbf{x}^p$ is a positive constant, it holds that

$$\mathrm{sgn}(\mathbf{x}^p\mathbf{W}) = \mathrm{sgn}\left(\mathbf{y}^p + \sum_{\ell \neq p}^{m} \mathbf{y}^\ell \frac{(\mathbf{x}^\ell \cdot \mathbf{x}^p)}{(\mathbf{x}^p \cdot \mathbf{x}^p)}\right).$$

To produce the output $\mathbf{y}^p$ the equation

$$\mathbf{y}^p = \mathrm{sgn}\left(\mathbf{y}^p + \sum_{\ell \neq p}^{m} \mathbf{y}^\ell \frac{(\mathbf{x}^\ell \cdot \mathbf{x}^p)}{(\mathbf{x}^p \cdot \mathbf{x}^p)}\right)$$

must hold. This is the case when the absolute value of all components of the perturbation term

$$\sum_{\ell \neq p}^{m} \mathbf{y}^\ell \frac{(\mathbf{x}^\ell \cdot \mathbf{x}^p)}{(\mathbf{x}^p \cdot \mathbf{x}^p)}$$

is smaller than 1. This means that the scalar products $\mathbf{x}^\ell \cdot \mathbf{x}^p$ must be small in comparison to the quadratic length of the vector $\mathbf{x}^p$ (equal to n for n-dimensional bipolar vectors). If randomly selected bipolar vectors are associated with other also randomly selected bipolar vectors, the probability is high that they will be nearly pairwise orthogonal, as long as not many of them are selected. The crosstalk will be small. In this case Hebbian learning will lead to an efficient set of weights for the associative network.

In the autoassociative case, in which a vector $\mathbf{x}^1$ is to be associated with itself, the matrix $\mathbf{W}^1$ can also be computed using Hebbian learning. The matrix is given by

$$\mathbf{W}^1 = (\mathbf{x}^1)^{\mathrm{T}}\mathbf{x}^1.$$

Some authors define the autocorrelation matrix $\mathbf{W}^1$ as $\mathbf{W}^1 = (1/n)(\mathbf{x}^1)^{\mathrm{T}}\mathbf{x}^1$. Since the additional positive constant does not change the sign of the excitation, we will not use it in our computations.

If a set of m row vectors $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^m$ is to be autoassociated, the weight matrix $\mathbf{W}$ is given by

$$\begin{aligned}\mathbf{W} &= (\mathbf{x}^1)^{\mathrm{T}}\mathbf{x}^1 + (\mathbf{x}^2)^{\mathrm{T}}\mathbf{x}^2 + \cdots + (\mathbf{x}^m)^{\mathrm{T}}\mathbf{x}^m \\ &= \mathbf{X}^{\mathrm{T}}\mathbf{X},\end{aligned}$$

where $\mathbf{X}$ is the $m \times n$ matrix whose rows are the m given vectors. The matrix $\mathbf{W}$ is now the autocorrelation matrix for the set of m given vectors. It is expected from $\mathbf{W}$ that it can lead to the reproduction of each one of the vectors $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^m$ when used as weight matrix of the autoassociative network. This means that

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

$$\operatorname{sgn}(\mathbf{x}^i\mathbf{W}) = \mathbf{x}^i \quad \text{for } i = 1, ..., m, \tag{12.4}$$

or alternatively

$$\operatorname{sgn}(\mathbf{XW}) = \mathbf{X}. \tag{12.5}$$

The function $\operatorname{sgn}(\mathbf{xW})$ can be considered to be a nonlinear operator. Equation (12.4) expresses the fact that the vectors $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^m$ are the eigenvectors of the nonlinear operator. The *learning problem for associative networks* consists in constructing the matrix $\mathbf{W}$ for which the nonlinear operator $\operatorname{sgn}(\mathbf{xW})$ has these eigenvectors as fixed points. This is just a generalization of the eigenvector concept to nonlinear functions. On the other hand, we do not want to simply use the identity matrix, since the objective of the whole exercise is to correct noisy input vectors. Those vectors located near to stored patterns should be attracted to them. For $\mathbf{W} = \mathbf{X}^{\mathrm{T}}\mathbf{X}$, (12.5) demands that

$$\operatorname{sgn}(\mathbf{XX}^{\mathrm{T}}\mathbf{X}) = \mathbf{X}. \tag{12.6}$$

If the vectors $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^m$ are pairwise orthogonal $\mathbf{X}^{\mathrm{T}}\mathbf{X}$ is the identity matrix multiplied by n and (12.6) is fulfilled. If the patterns are nearly pairwise orthogonal $\mathbf{X}^{\mathrm{T}}\mathbf{X}$ is nearly the identity matrix (times n) and the associative network continues to work.

12.2.2 Geometric interpretation of Hebbian learning

The matrices $\mathbf{W}^1, \mathbf{W}^2, \ldots, \mathbf{W}^m$ in equation (12.3) can be interpreted geometrically. This would provide us with a hint as to the possible ways of improving the learning method. Consider first the matrix $\mathbf{W}^1$, defined in the autoassociative case as

$$\mathbf{W}^1 = (\mathbf{x}^1)^{\mathrm{T}}\mathbf{x}^1.$$

Any input vector $\mathbf{z}$ fed to the network is *projected* into the linear subspace L_1 spanned by the vector $\mathbf{x}^1$, since

$$\mathbf{zW}^1 = \mathbf{z}(\mathbf{x}^1)^{\mathrm{T}}\mathbf{x}^1 = (\mathbf{z}\,(\mathbf{x}^1)^{\mathrm{T}})\mathbf{x}^1 = c_1\,\mathbf{x}^1,$$

where c_1 represents a constant. The matrix $\mathbf{W}^1$ represents, in general, a non-orthogonal projection operator that projects the vector $\mathbf{z}$ in L_1. A similar interpretation can be derived for each weight matrix $\mathbf{W}^2$ to $\mathbf{W}^m$. The matrix $\mathbf{W} = \sum_{i=1}^m \mathbf{W}^i$ is therefore a linear transformation which projects a vector $\mathbf{z}$ into the linear subspace spanned by the vectors $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^m$, since

$$\begin{aligned}\mathbf{zW} &= \mathbf{zW}^1 + \mathbf{zW}^2 + \cdots + \mathbf{zW}^m \\ &= c_1\mathbf{x}^1 + c_2\mathbf{x}^2 + \cdots + c_m\mathbf{x}^m\end{aligned}$$

with constants $c_1, c_2, \ldots, c_m$. In general $\mathbf{W}$ does not represent an orthogonal projection.

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

12.2.3 Networks as dynamical systems – some experiments

How good is Hebbian learning when applied to an associative network? Some empirical results can illustrate this point, as we proceed to show in this section.

Since associative networks can be considered to be dynamical systems, one of the important issues is how to identify their attractors and how extended the basins of attraction are, that is, how large is the region of input space mapped to each one of the fixed points of the system. We have already discussed how to engineer a nonlinear operator whose eigenvectors are the patterns to be stored. Now we investigate this matter further and measure the changes in the basins of attraction when the patterns are learned one after the other using the Hebb rule. In order to measure the size of the basins of attraction we must use a suitable metric. This will be the *Hamming distance* between bipolar vectors, which is just the number of different components which both contain.

Table 12.1. Percentage of 10-dimensional vectors with a Hamming distance (H) from 0 to 4, which converge to a stored vector in a single iteration. The number of stored vectors increases from 1 to 10.

	Number of stored vectors (dimension 10)									
H	1	2	3	4	5	6	7	8	9	10
0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
1	100.0	100.0	90.0	85.0	60.0	60.0	54.3	56.2	45.5	33.0
2	100.0	86.7	64.4	57.2	40.0	31.8	22.5	23.1	17.0	13.3
3	100.0	50.0	38.6	25.4	13.5	8.3	4.8	5.9	3.1	2.4
4	100.0	0.0	9.7	7.4	4.5	2.7	0.9	0.8	0.3	0.2

Table 12.1 shows the results of a computer experiment. Ten randomly chosen vectors in 10-dimensional space were stored in the weight matrix of an associative network using Hebbian learning. The weight matrix was computed iteratively, first for one vector, then for two, etc. After each update of the weight matrix the number of vectors with a Hamming distance from 0 to 4 to the stored vectors, and which could be mapped to each one of them by the network, was counted. The table shows the percentage of vectors that were mapped in a single iteration to a stored pattern. As can be seen, each new stored pattern reduces the size of the average basins of attraction, until by 10 stored vectors only 33% of the vectors with a single different component are mapped to one of them.

The table shows the average results for all stored patterns. When only one vector has been stored, all vectors up to Hamming distance 4 converge to it. If two vectors have been stored, only 86.7% of the vectors with Hamming

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

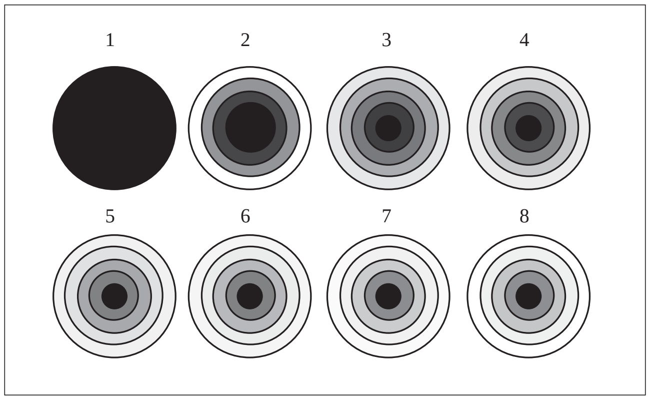


Fig. 12.5. The grey shading of each concentric circle represents the percentage of vectors with Hamming distance from 0 to 4 to stored patterns and which converge to them. The smallest circle represents the vectors with Hamming distance 0. Increasing the number of stored patterns from 1 to 8 reduces the size of the basins of attraction.

distance 2 converge to the stored patterns. Figure 12.5 is a graphical representation of the changes to the basins of attraction after each new pattern has been stored. The basins of attraction become continuously smaller and less dense.

Table 12.1 shows only the results for vectors with a maximal Hamming distance of 4, because vectors with a Hamming distance greater than 5 are mapped not to $\mathbf{x}$ but to $-\mathbf{x}$, when $\mathbf{x}$ is one of the stored patterns. This is so because

$$\text{sgn}(-\mathbf{x}\mathbf{W}) = -\text{sgn}(\mathbf{x}\mathbf{W}) = -\mathbf{x}.$$

These additional stored patterns are usually a nuisance. They are called *spurious* stable states. However, they are inevitable in the kind of associative networks considered in this chapter.

The results of Table 12.1 can be improved using a recurrent network of the type shown in Figure 12.3. The operator $\text{sgn}(\mathbf{x}\mathbf{W})$ is used repetitively. If an input vector does not converge to a stored pattern in one iteration, its image is nevertheless rotated in the direction of the fixed point. An additional iteration can then lead to convergence.

The iterative method was described before for the eigenvector automaton. The disadvantage of that automaton was that a single vector attracts almost the whole of input space. The nonlinear operator $\text{sgn}(\mathbf{x}\mathbf{W})$ provides a solution for this problem. On the one hand, the stored vectors can be reproduced, i.e., $\text{sgn}(\mathbf{x}\mathbf{W}) = \mathbf{x}$. There are no "eigenvalues" different from 1. On the other hand, the nonlinear sign function does not allow a single vector to attract most of input space. Input space is divided more regularly into basins of attraction for the different stored vectors. Table 12.2 shows the convergence

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

of an autoassociative network when five iterations are used. Only the first seven vectors used in Table 12.1 were considered again for this table.

Table 12.2. Percentage of 10-dimensional vectors with a Hamming distance (H) from 0 to 4, which converge to a stored vector in five iterations. The number of stored vectors increases from 1 to 7.

	Number of stored vectors (dimension 10)						
H	1	2	3	4	5	6	7
0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
1	100.0	100.0	90.0	85.0	60.0	60.0	54.3
2	100.0	100.0	72.6	71.1	41.8	34.8	27.9
3	100.0	80.0	48.6	47.5	18.3	10.8	8.5
4	100.0	42.8	21.9	22.3	6.8	3.7	2.0

The table shows an improvement in the convergence properties of the network, that is, an expansion of the average size and density of the basins of attraction. After six stored vectors the results of Table 12.2 become very similar to those of Table 12.1.

The results of Table 12.1 and Table 12.2 can be compared graphically. Figure 12.6 shows the percentage of vectors with a Hamming distance of 0 to 5 from a stored vector that converge to it. The broken line shows the results for the recurrent network of Table 12.2. The continuous line summarizes the results for a single iteration. Comparison of the "single shot" method and the recurrent computation show that the latter is more effective at least as long as not too many vectors have been stored. When the *capacity* of the weight matrix begins to be put under strain, the differences between the two methods disappear.

The sizes of the basins of attraction in the feed-forward and the recurrent network can be compared using an index I for their size. This index can be defined as

$$I = \sum_{h=0}^{5} h p_h$$

where p_h represents the percentage of vectors with Hamming distance h from a stored vector which converge to it. Two indices were computed using the data in Table 12.1 and Table 12.2. The results are shown in Figure 12.7. Both indices are clearly different up to five stored vectors.

As all these comparisons show, the size of the basins of attraction falls very fast. This can be explained by remembering that five stored vectors imply at least ten actually stored patterns, since together with $\mathbf{x}$ the vector $-\mathbf{x}$ is also stored. Additionally, distortion of the basins of attraction produced by

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

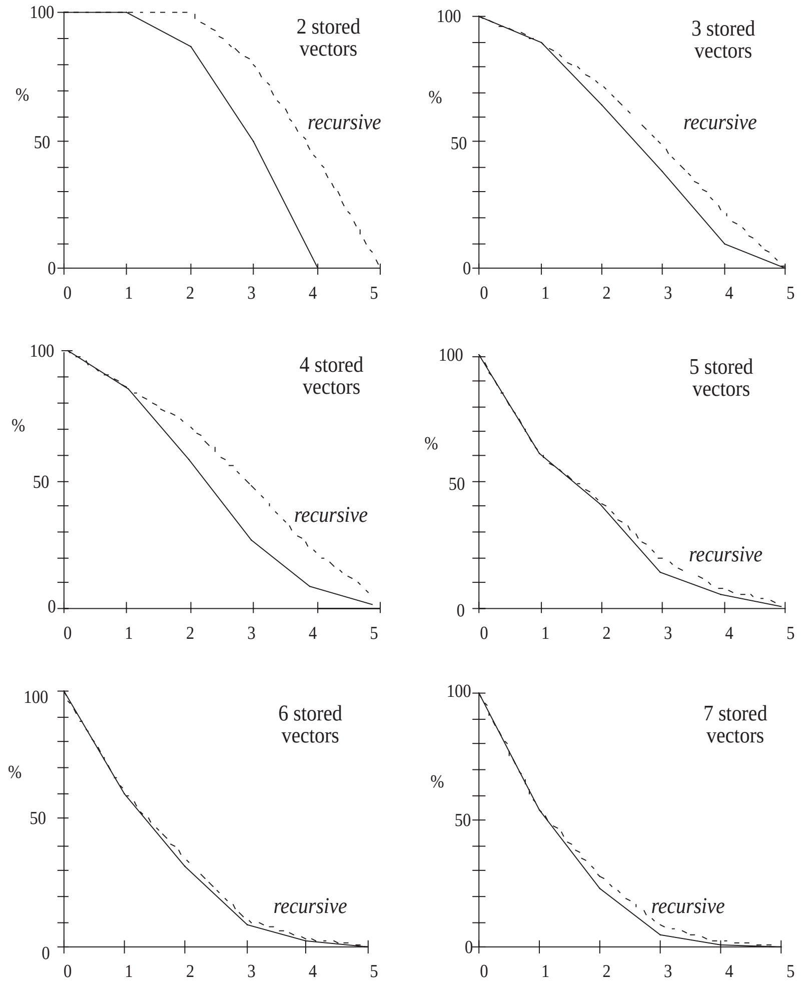


Fig. 12.6. Comparison of the percentage of vectors with a Hamming distance from 0 to 5 from stored patterns and which converge to them

Hebbian learning leads to the appearance of other spurious stable states. A computer experiment was used to count the number of spurious states that appear when 2 to 7 random bipolar vectors are stored in a 10×10 associative matrix. Table 12.3 shows the fast increase in the number of spurious states.

Spurious states other than the negative stored patterns appear because the crosstalk becomes too large. An alternative way to minimize the crosstalk term is to use another kind of learning rule, as we discuss in the next section.

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

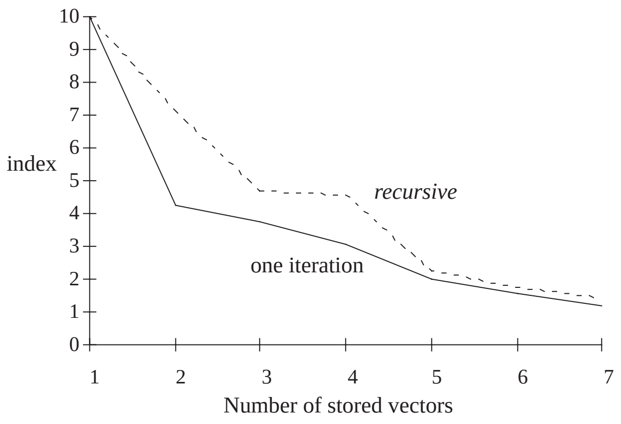


Fig. 12.7. Comparison of the indices of attraction for an associative network with and without feedback

Table 12.3. Increase in the number of spurious states when the number of stored patterns increases from 2 to 7 (dimension 10)

stored vectors	2	3	4	5	6	7
negative vectors	2	3	4	5	6	7
spurious fixed points	2	4	8	16	24	45
total	4	7	12	21	30	52

12.2.4 Another visualization

A second kind of visualization of the basins of attraction allows us to perceive their gradual fragmentation in an associative network. Figure 12.8 shows the result of an experiment using input vectors in a 100-dimensional space. Several vectors were stored and the size of the basins of attraction of one of them and its bipolar complement were monitored using the following technique: the 100 components of each vector were divided into two groups of 50 components using random selection. The Hamming distance of a vector to the stored vector was measured by counting the number of different components in each group. In this way the vector $\mathbf{z}$ with Hamming distances 20 and 30 to the vector $\mathbf{x}$, according to the chosen partition in two groups of components, is a vector with total Hamming distance 50 to the stored vector. The chosen partition allows us to represent the vector $\mathbf{z}$ as a point at the position $(20, 30)$ in a two-dimensional grid. This point is colored black if $\mathbf{z}$ is associated with $\mathbf{x}$ by the associative network, otherwise it is left white. For each point in the grid a vector with the corresponding Hamming distance to $\mathbf{x}$ was generated.

As Figure 12.8 shows, when only 4 vectors have been stored, the vector $\mathbf{x}$ and its complement $-\mathbf{x}$ attract a considerable portion of their neighborhood.

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

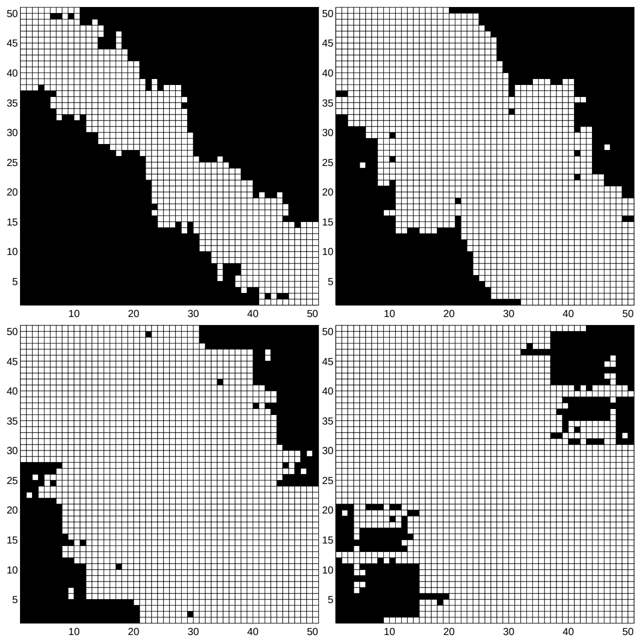


Fig. 12.8. Basin of attraction in 100-dimensional space of a stored vector. The number of stored patterns is 4, 6, 10 and 15, from top to bottom, left to right.

As the number of stored vectors increases, the basins of attraction become ragged and smaller, until by 15 stored vectors they are starting to become unusable. This is near to the theoretical limit for the capacity of an associative network.

12.3 The capacity problem

The experiments of the previous sections have shown that the basins of attraction of stored patterns deteriorate every time new vectors are presented to an associative network. If the crosstalk term becomes too large, it can even happen that previously stored patterns are lost, because when they are presented to the network one or more of their bits are flipped by the associative computation. We would like to keep the probability that this could happen low, so that stored patterns can always be recalled. Depending on the upper bound that we want to put on this probability some limits to the number of patterns m that can be stored safely in an autoassociative network with an $n \times n$ weight matrix $\mathbf{W}$ will arise. This is called the maximum *capacity* of the network and an often quoted rule of thumb is $m \approx 0.18n$. It is actually easy to derive this rule [189].

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

The crosstalk term for n-dimensional bipolar vectors and m patterns in the autoassociative case is

$$\frac{1}{n}\sum_{\ell \neq p}^{m} \mathbf{x}^{\ell}(\mathbf{x}^{\ell} \cdot \mathbf{x}^{p}).$$

This can flip a bit of a stored pattern if the magnitude of this term is larger than 1 and if it is of opposite sign. Assume that the stored vectors are chosen randomly from input space. In this case the crosstalk term for bit i of the input vector is given by

$$\frac{1}{n}\sum_{\ell \neq p}^{m} x_i^{\ell}(\mathbf{x}^{\ell} \cdot \mathbf{x}^{p}).$$

This is a sum of $(m-1)n$ bipolar bits. Since the components of each pattern have been selected randomly we can think of mn random bit selections (for large m and n). The expected value of the sum is zero. It can be shown that the sum has a binomial distribution and for large mn we can approximate it with a normal distribution. The standard deviation of the distribution is $\sigma = \sqrt{m/n}$. The probability of error P that the sum becomes larger than 1 or -1 is given by the area under the Gaussian from 1 to ∞ or from -1 to $-\infty$. That is,

$$P = \frac{1}{\sqrt{2\pi}\sigma}\int_{1}^{\infty} \mathrm{e}^{-x^2/(2\sigma^2)}dx$$

If we set the upper bound for one bit failure at 0.01, the above expression can be solved numerically to find the appropriate combination of m and n. The numerical solution leads to $m \approx 0.18n$ as mentioned before.

Note that these are just probabilistic results. If the patterns are correlated, even $m < 0.18n$ can produce problems. Also, we did not consider the case of a recursive computation and only considered the case of a false bit. A more precise computation does not essentially change the order of magnitude of these results: an associative network remains statistically safe only if fewer than $0.18n$ patterns are stored. If we demand perfect recall of all patterns more stringent bounds are needed [189]. The experiments in the previous section are on such a small scale that no problems were found with the recall of stored patterns even when m was much larger than $0.18n$.

12.4 The pseudoinverse

Hebbian learning produces good results when the stored patterns are nearly orthogonal. This is the case when m bipolar vectors are selected randomly from an n-dimensional space, n is large enough and m much smaller than n. In real applications the patterns are almost always correlated and the crosstalk in the expression

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

$$\mathbf{x}^p\mathbf{W} = \mathbf{y}^p(\mathbf{x}^p \cdot \mathbf{x}^p) + \sum_{\ell \neq p}^{m} \mathbf{y}^\ell(\mathbf{x}^\ell \cdot \mathbf{x}^p)$$

affects the recall process because the scalar products $\mathbf{x}^\ell \cdot \mathbf{x}^p$, for $\ell \neq p$, are not small enough. This causes a reduction in the *capacity* of the associative network, that is, of the number of patterns that can be stored and later recalled. Consider the example of scanned letters of the alphabet, digitized using a 16×16 grid of pixels. The pattern vectors do not occupy input space homogeneously but concentrate around a small region. We must therefore look for alternative learning methods capable of minimizing the crosstalk between the stored patterns. One of the preferred methods is using the pseudoinverse of the pattern matrix instead of the correlation matrix.

12.4.1 Definition and properties of the pseudoinverse

Let $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^m$ be n-dimensional vectors and associate them with the m k-dimensional vectors $\mathbf{y}^1, \mathbf{y}^2, \ldots, \mathbf{y}^m$. The matrix $\mathbf{X}$ is, as before, the $m \times n$ matrix whose rows are the vectors $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^m$. The rows of the $m \times k$ matrix $\mathbf{Y}$ are the vectors $\mathbf{y}^1, \mathbf{y}^2, \ldots, \mathbf{y}^m$. We are looking for a weight matrix $\mathbf{W}$ such that

$$\mathbf{XW} = \mathbf{Y}.$$

Since in general $m \neq n$ and the vectors $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^m$ are not necessarily linearly independent, the matrix $\mathbf{X}$ cannot be inverted. However, we can look for a matrix $\mathbf{W}$ which minimizes the quadratic norm of the matrix $\mathbf{XW} - \mathbf{Y}$, that is, the sum of the squares of all its elements. It is a well-known result of linear algebra that the expression $\|\mathbf{XW} - \mathbf{Y}\|^2$ is minimized by the matrix $\mathbf{W} = \mathbf{X}^+\mathbf{Y}$, where $\mathbf{X}^+$ is the so-called pseudoinverse of the matrix $\mathbf{X}$. In some sense the pseudoinverse is the best "approximation" to an inverse that we can get, and if $\mathbf{X}^{-1}$ exists, then $\mathbf{X}^{-1} = \mathbf{X}^+$.

Definition 14. *The pseudoinverse of a real $m \times n$ matrix is the real matrix $\mathbf{X}^+$ with the following properties:*

a) $\mathbf{X}\mathbf{X}^+\mathbf{X} = \mathbf{X}$.
b) $\mathbf{X}^+\mathbf{X}\mathbf{X}^+ = \mathbf{X}^+$.
c) $\mathbf{X}^+\mathbf{X}$ *and* $\mathbf{X}\mathbf{X}^+$ *are symmetrical.*

It can be shown that the pseudoinverse of a matrix $\mathbf{X}$ always exists and is unique [14]. We can now prove the result mentioned before [185].

Proposition 18. *Let $\mathbf{X}$ be an $m \times n$ real matrix and $\mathbf{Y}$ be an $m \times k$ real matrix. The $n \times k$ matrix $\mathbf{W} = \mathbf{X}^+\mathbf{Y}$ minimizes the quadratic norm of the matrix $\mathbf{XW} - \mathbf{Y}$.*

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

Proof. Define $E = \|\mathbf{XW} - \mathbf{Y}\|^2$. The value of E can be computed from the equation

$$E = \mathrm{tr}(\mathbf{S}),$$

where $\mathbf{S} = (\mathbf{XW} - \mathbf{Y})^{\mathrm{T}}(\mathbf{XW} - \mathbf{Y})$ and $\mathrm{tr}(\cdot)$ denotes the function that computes the trace of a matrix, that is, the sum of all its diagonal elements. The matrix $\mathbf{S}$ can be rewritten as

$$\mathbf{S} = (\mathbf{X}^+\mathbf{Y} - \mathbf{W})^{\mathrm{T}}\mathbf{X}^{\mathrm{T}}\mathbf{X}(\mathbf{X}^+\mathbf{Y} - \mathbf{W}) + \mathbf{Y}^{\mathrm{T}}(\mathbf{I} - \mathbf{XX}^+)\mathbf{Y}. \tag{12.7}$$

This can be verified by bringing $\mathbf{S}$ back to its original form. Equation (12.7) can be written as

$$\mathbf{S} = (\mathbf{X}^+\mathbf{Y} - \mathbf{W})^{\mathrm{T}}(\mathbf{X}^{\mathrm{T}}\mathbf{XX}^+\mathbf{Y} - \mathbf{X}^{\mathrm{T}}\mathbf{XW}) + \mathbf{Y}^{\mathrm{T}}(\mathbf{I} - \mathbf{XX}^+)\mathbf{Y}.$$

Since the matrix $\mathbf{XX}^+$ is symmetrical (from the definition of $\mathbf{X}^+$), the above expression transforms to

$$\mathbf{S} = (\mathbf{X}^+\mathbf{Y} - \mathbf{W})^{\mathrm{T}}((\mathbf{XX}^+\mathbf{X})^{\mathrm{T}}\mathbf{Y} - \mathbf{X}^{\mathrm{T}}\mathbf{XW}) + \mathbf{Y}^{\mathrm{T}}(\mathbf{I} - \mathbf{XX}^+)\mathbf{Y}.$$

Since from the definition of $\mathbf{X}^+$ we know that $\mathbf{XX}^+\mathbf{X} = \mathbf{X}$ it follows that

$$\begin{aligned}
\mathbf{S} &= (\mathbf{X}^+\mathbf{Y} - \mathbf{W})^{\mathrm{T}}(\mathbf{X}^{\mathrm{T}}\mathbf{Y} - \mathbf{X}^{\mathrm{T}}\mathbf{XW}) + \mathbf{Y}^{\mathrm{T}}(\mathbf{I} - \mathbf{XX}^+)\mathbf{Y} \\
&= (\mathbf{X}^+\mathbf{Y} - \mathbf{W})^{\mathrm{T}}\mathbf{X}^{\mathrm{T}}(\mathbf{Y} - \mathbf{XW}) + \mathbf{Y}^{\mathrm{T}}(\mathbf{I} - \mathbf{XX}^+)\mathbf{Y} \\
&= (\mathbf{XX}^+\mathbf{Y} - \mathbf{XW})^{\mathrm{T}}(\mathbf{Y} - \mathbf{XW}) + \mathbf{Y}^{\mathrm{T}}(\mathbf{I} - \mathbf{XX}^+)\mathbf{Y} \\
&= (-\mathbf{XW})^{\mathrm{T}}(\mathbf{Y} - \mathbf{XW}) + \mathbf{Y}^{\mathrm{T}}\mathbf{XX}^+(\mathbf{Y} - \mathbf{XW}) + \mathbf{Y}^{\mathrm{T}}(\mathbf{I} - \mathbf{XX}^+)\mathbf{Y} \\
&= (-\mathbf{XW})^{\mathrm{T}}(\mathbf{Y} - \mathbf{XW}) + \mathbf{Y}^{\mathrm{T}}(-\mathbf{XW}) + \mathbf{Y}^{\mathrm{T}}\mathbf{Y} \\
&= (\mathbf{Y} - \mathbf{XW})^{\mathrm{T}}(\mathbf{Y} - \mathbf{XW}).
\end{aligned}$$

This confirms that equation (12.7) is correct. Using (12.7) the value of E can be written as

$$E = \mathrm{tr}\left((\mathbf{X}^+\mathbf{Y} - \mathbf{W})^{\mathrm{T}}\mathbf{X}^{\mathrm{T}}\mathbf{X}(\mathbf{X}^+\mathbf{Y} - \mathbf{W})\right) + \mathrm{tr}\left(\mathbf{Y}^{\mathrm{T}}(\mathbf{I} - \mathbf{XX}^+)\mathbf{Y}\right),$$

since the trace of an addition of two matrices is the addition of the trace of each matrix. The trace of the second matrix is a constant. The trace of the first is positive or zero, since the trace of a matrix of the form $\mathbf{AA}^{\mathrm{T}}$ is always positive or zero. The minimum of E is reached therefore when $\mathbf{W} = \mathbf{X}^+\mathbf{Y}$, as we wanted to prove. □

An interesting corollary of the above proposition is that the pseudoinverse of a matrix $\mathbf{X}$ minimizes the norm of the matrix $\mathbf{XX}^+ - \mathbf{I}$. This is what we mean when we say that the pseudoinverse is the second best choice if the inverse of a matrix is not defined.

The quadratic norm E has a straightforward interpretation for our associative learning task. Minimizing $E = \|\mathbf{XW} - \mathbf{Y}\|^2$ amounts to minimizing

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

the sum of the quadratic norm of the vectors $\mathbf{x}^i\mathbf{W} - \mathbf{y}^i$, where $(\mathbf{x}^i, \mathbf{y}^i)$, for $i = 1, \ldots, m$, are the vector pairs we want to associate. The identity

$$\mathbf{x}^i\mathbf{W} = \mathbf{y}^i + (\mathbf{x}^i\mathbf{W} - \mathbf{y}^i)$$

always holds. Since the desired result of the computation is $\mathbf{y}^i$, the term $\mathbf{x}^i\mathbf{W} - \mathbf{y}^i$ represents the deviation from the ideal situation, that is, the crosstalk in the associative network. The quadratic norm E defined before can also be written as

$$E = \sum_{i=1}^{m} \left\| \mathbf{x}^i\mathbf{W} - \mathbf{y}^i \right\|^2 .$$

Minimizing E amounts to minimizing the deviation from perfect recall. The pseudoinverse can thus be used to compute an optimal weight matrix $\mathbf{W}$.

Table 12.4. Percentage of 10-dimensional vectors with a Hamming distance (H) from 0 to 4, which converge to a stored vector in five iterations. The number of stored vectors increases from 1 to 7. The weight matrix is $\mathbf{X}^+\mathbf{X}$.

Number of stored vectors (dimension 10)

H	1	2	3	4	5	6	7
0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
1	100.0	100.0	90.0	85.0	50.0	50.0	30.0
2	100.0	86.7	77.8	60.0	22.7	10.4	1.3
3	100.0	50.8	40.8	30.4	3.2	0.0	0.0
4	100.0	1.9	7.6	8.6	0.0	0.0	0.0

An experiment similar to the ones performed before (Table 12.4) shows that using the pseudoinverse to compute the weight matrix produces better results than when the correlation matrix is used. Only one iteration was performed during the associative recall. The table also shows that if more than five vectors are stored the quality of the results does not improve. This can be explained by noting that the capacity of the associative network has been overflowed. The stored vectors were also randomly chosen. The pseudoinverse method performs better than Hebbian learning when the patterns are correlated.

12.4.2 Orthogonal projections

In Sect. 12.2.2 we provided a geometric interpretation of the effect of the correlation matrix in the associative computation. We arrived at the conclusion that when an n-dimensional vector $\mathbf{x}^1$ is autoassociated, the weight matrix $(\mathbf{x}^1)^{\mathrm{T}}\mathbf{x}^1$ projects the whole of input space into the linear subspace spanned by

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

$\mathbf{x}^1$. However, the projection is not an orthogonal projection. The pseudoinverse can be used to construct an operator capable of projecting the input vector orthogonally into the subspace spanned by the stored patterns.

What is the advantage of performing an orthogonal projection? An autoassociative network must associate each input vector with the nearest stored pattern and must produce this as the result. In nonlinear associative networks this is done in two steps: the input vector $\mathbf{x}$ is projected first onto the linear subspace spanned by the stored patterns, and then the nonlinear function sgn$(\cdot)$ is used to find the bipolar vector nearest to the projection. If the projection falsifies the information about the distance of the input to the stored patterns, then a suboptimal selection could be the result. Figure 12.9 illustrates this problem. The vector $\mathbf{x}$ is projected orthogonally and non-orthogonally in the space spanned by the vectors $\mathbf{x}^1$ and $\mathbf{x}^2$. The vector $\mathbf{x}$ is closer to $\mathbf{x}^2$ than to $\mathbf{x}^1$. The orthogonal projection $\tilde{\mathbf{x}}$ is also closer to $\mathbf{x}^2$. However, the non-orthogonal projection $\breve{\mathbf{x}}$ is closer to $\mathbf{x}^1$ as to $\mathbf{x}^2$. This is certainly a scenario we should try to avoid.

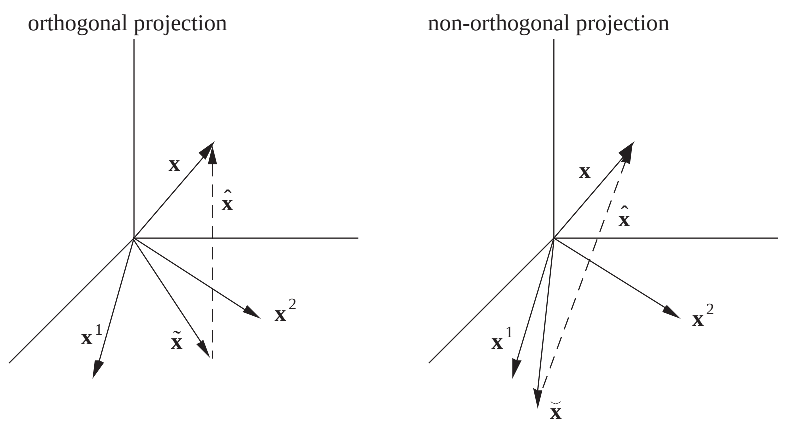


Fig. 12.9. Projections of a vector $\mathbf{x}$

In what follows we consider only the orthogonal projection. If we use the scalar product as the metric to assess the distance between patterns, the distance between the input vector $\mathbf{x}$ and the stored vector $\mathbf{x}^i$ is given by

$$d = \mathbf{x} \cdot \mathbf{x}^i = (\tilde{\mathbf{x}} + \hat{\mathbf{x}}) \cdot \mathbf{x}^i = \tilde{\mathbf{x}} \cdot \mathbf{x}^i, \qquad (12.8)$$

where $\tilde{\mathbf{x}}$ represents the orthogonal projection onto the vector subspace spanned by the stored vectors, and $\hat{\mathbf{x}} = \mathbf{x} - \tilde{\mathbf{x}}$. Equation (12.8) shows that the original distance d is equal to the distance between the projection $\tilde{\mathbf{x}}$ and the vector $\mathbf{x}^i$. The orthogonal projection does not falsify the original information. We must therefore only show that the pseudoinverse provides a simple way

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

to find the orthogonal projection to the linear subspace defined by the stored patterns.

Consider m n-dimensional vectors $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^m$, where $m < n$. Let $\tilde{\mathbf{x}}$ be the orthogonal projection of an arbitrary input vector $\mathbf{x} \neq \mathbf{0}$ on the subspace spanned by the m given vectors. In this case

$$\mathbf{x} = \tilde{\mathbf{x}} + \hat{\mathbf{x}},$$

where $\hat{\mathbf{x}}$ is orthogonal to the vectors $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^m$. We want to find an $n \times n$ matrix $\mathbf{W}$ such that

$$\mathbf{x}\mathbf{W} = \tilde{\mathbf{x}}.$$

Since $\hat{\mathbf{x}} = \mathbf{x} - \tilde{\mathbf{x}}$, the matrix $\mathbf{W}$ must fulfill

$$\hat{\mathbf{x}} = \mathbf{x}(\mathbf{I} - \mathbf{W}). \tag{12.9}$$

Let $\mathbf{X}$ be the $m \times n$ matrix whose rows are the vectors $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^m$. Then it holds for $\hat{\mathbf{x}}$ that

$$\mathbf{X}\hat{\mathbf{x}}^{\mathrm{T}} = \mathbf{0},$$

since $\hat{\mathbf{x}}$ is orthogonal to all vectors $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^m$. A possible solution for this equation is

$$\hat{\mathbf{x}}^{\mathrm{T}} = (\mathbf{I} - \mathbf{X}^+\mathbf{X})\mathbf{x}^{\mathrm{T}}, \tag{12.10}$$

because in this case

$$\mathbf{X}\hat{\mathbf{x}}^{\mathrm{T}} = \mathbf{X}(\mathbf{I} - \mathbf{X}^+\mathbf{X})\mathbf{x}^{\mathrm{T}} = (\mathbf{X} - \mathbf{X})\mathbf{x}^{\mathrm{T}} = \mathbf{0}.$$

Since the matrix $\mathbf{I} - \mathbf{X}^+\mathbf{X}$ is symmetrical it holds that

$$\hat{\mathbf{x}} = \mathbf{x}(\mathbf{I} - \mathbf{X}^+\mathbf{X}). \tag{12.11}$$

A direct comparison of equations (12.9) and (12.11) shows that $\mathbf{W} = \mathbf{X}^+\mathbf{X}$, since the orthogonal projection $\hat{\mathbf{x}}$ is unique. This amounts to a proof that the orthogonal projection of a vector $\mathbf{x}$ can be computed by multiplying it with the matrix $\mathbf{X}^+\mathbf{X}$.

What happens if the number of patterns m is larger than the dimension of the input space? Assume that the m n-dimensional vectors $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^m$ are to be associated with the m real numbers $y_1, y_2, \ldots, y_m$. Let $\mathbf{X}$ be the matrix whose rows are the given patterns and let $\mathbf{y} = (y_1, y_2, \ldots, y_m)$. We are looking for the $n \times 1$ matrix $\mathbf{W}$ such that

$$\mathbf{X}\mathbf{W} = \mathbf{y}^{\mathrm{T}}.$$

In general there is no exact solution for this system of m equations in n variables. The best approximation is given by $\mathbf{W} = \mathbf{X}^+\mathbf{y}^{\mathrm{T}}$, which minimizes the quadratic error $\|\mathbf{X}\mathbf{W} - \mathbf{y}^{\mathrm{T}}\|^2$. The solution $\mathbf{X}^+\mathbf{X}^{\mathrm{T}}$ is therefore the same as that we find if we use *linear regression.*

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

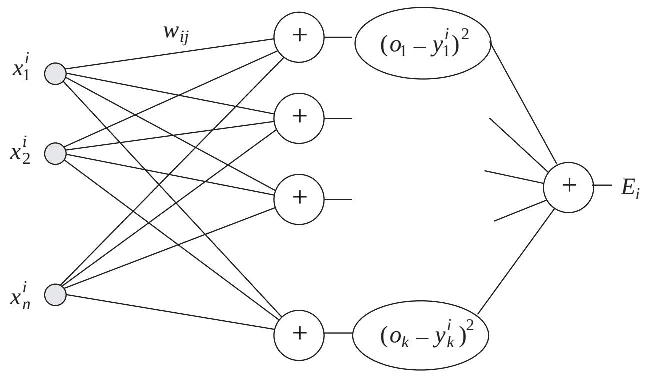


Fig. 12.10. Backpropagation network for a linear associative memory

The only open question is how best to compute the pseudoinverse. We already discussed in Chap. 9 that there are several algorithms which can be used, but that a simple approach is to compute an approximation using a backpropagation network. A network, such as the one shown in Figure 12.10, can be used to find the appropriate weights for an associative memory. The units in the first layer are linear associators. The learning task consists in finding the weight matrix $\mathbf{W}$ with elements w_{ij} that produces the best mapping from the vectors $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^m$ to the vectors $\mathbf{y}^1, \mathbf{y}^2, \ldots, \mathbf{y}^m$. The network is extended as shown in Figure 12.10. For the i-th input vector, the output of the network is compared to the vector $\mathbf{y}^i$ and the quadratic error E_i is computed. The total quadratic error $E = \sum_{i=1}^m E_i$ is the quadratic norm of the matrix $\mathbf{XW} - \mathbf{Y}$. Backpropagation can find the matrix $\mathbf{W}$ which minimizes the expression $\|\mathbf{XW} - \mathbf{Y}\|^2$. If the input vectors are linearly independent and $m < n$, then a solution with zero error can be found. If $m \geq n$ and $\mathbf{Y} = \mathbf{I}$, the network of Figure 12.10 can be used to find the pseudoinverse $\mathbf{X}^+$ or the inverse $\mathbf{X}^{-1}$ (if it exists). This algorithm for the computation of the pseudoinverse brings backpropagation networks in direct contact with the problem of associative networks. Strictly speaking, we should minimize $\|\mathbf{XW} - \mathbf{Y}\|^2$ and the quadratic norm of $\mathbf{W}$. This is equivalent to introducing a *weight decay term* in the backpropagation computation.

12.4.3 Holographic memories

Associative networks have found many applications for pattern recognition tasks. Figure 12.11 shows the "canonical" example. The faces were scanned and encoded as vectors. A white pixel was encoded as -1 and a black pixel as 1. Some images were stored in an associative network. When later a *key*, that is, an incomplete or noisy version of one of the stored images is presented to the network, this completes the image and finds the one with the greatest similarity. This is called an *associative recall*.

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

Fig. 12.11. Associative recall of stored patterns

This kind of application can be implemented in conventional workstations. If the dimension of the images becomes too large (for example 10^6 pixels for a 1000×1000 image, the only alternative is to use innovative computer architectures, like so-called *holographic memories*. They work with similar methods to those described here, but computation is performed in parallel using optical elements. We leave the discussion of these architectures for Chap. 18.

12.4.4 Translation invariant pattern recognition

It would be nice if the process of associative recall could be made robust in the sense that those patterns which are very similar, except for a translation in the plane, could be identified as being, in fact, similar. This can be done using the two-dimensional Fourier transform of the scanned images. Figure 12.12 shows an example of two identical patterns positioned with a small displacement from each other.

Fig. 12.12. The pattern “0” at two positions of a grid

The two-dimensional discrete Fourier transform of a two-dimensional array is computed by first performing a one-dimensional Fourier transform of the rows of the array and then a one-dimensional Fourier transform of the columns of the results (or vice versa). It is easy to show that the absolute value of the Fourier coefficients does not change under a translation of the two-dimensional pattern. Since the Fourier transform also has the property that it preserves angles, that is, similar patterns in the original domain are also similar in

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

the Fourier domain, this preprocessing can be used to implement translation-invariant associative recall. In this case the vectors which are stored in the network are the absolute values of the Fourier coefficients for each pattern.

Definition 15. *Let $a(x,y)$ denote an array of real values, where x and y represent integers such that $0 \leq x \leq N-1$ and $0 \leq y \leq N-1$. The two-dimensional discrete Fourier transform $F_a(f_x,f_y)$ of $a(x,y)$ is given by*

$$F_a(f_x,f_y) = \frac{1}{N}\sum_{y=0}^{N-1}\sum_{x=0}^{N-1} a(x,y)\mathrm{e}^{\frac{2\pi}{N}ixf_x}\mathrm{e}^{\frac{2\pi}{N}iyf_y}$$

where $0 \leq f_x \leq N-1$ and $0 \leq f_y \leq N-1$.

Consider two arrays $a(x,y)$ and $b(x,y)$ of real values. Assume that they represent the same pattern, but with a certain displacement, that is $b(x,y) = a(x+d_x, y+d_y)$, where d_x and d_y are two given integers. Note that the additions $x+d_x$ and $y+d_y$ are performed modulo N (that is, the displaced patterns wrap around the two-dimensional array). The two-dimensional Fourier transform of the array $b(x,y)$ is therefore

$$F_b(f_x,f_y) = \frac{1}{N}\sum_{y=0}^{N-1}\sum_{x=0}^{N-1} a(x+d_x,y+d_y)\mathrm{e}^{\frac{2\pi}{N}ixf_x}\mathrm{e}^{\frac{2\pi}{N}iyf_y}.$$

With the change of indices $x' = (x+d_x) \bmod N$ and $y' = (y+d_y) \bmod N$ the above expression transforms to

$$F_b(f_x,f_y) = \frac{1}{N}\sum_{y=0}^{N-1}\sum_{x=0}^{N-1} a(x',y')\mathrm{e}^{\frac{2\pi}{N}ix'f_x}\mathrm{e}^{\frac{2\pi}{N}iy'f_y}\mathrm{e}^{\frac{-2\pi}{N}id_xf_x}\mathrm{e}^{\frac{-2\pi}{N}id_yf_y}.$$

The two sums can be rearranged and the constant factors taken out of the double sum to yield

$$F_b(f_x,f_y) = \frac{1}{N}\mathrm{e}^{\frac{-2\pi}{N}id_xf_x}\mathrm{e}^{\frac{-2\pi}{N}id_yf_y}\sum_{y'=0}^{N-1}\sum_{x'=0}^{N-1} a(x',y')\mathrm{e}^{\frac{2\pi}{N}ix'f_x}\mathrm{e}^{\frac{2\pi}{N}iy'f_y}.$$

which can be written as

$$F_b(f_x,f_y) = \mathrm{e}^{\frac{-2\pi}{N}id_xf_x}\mathrm{e}^{\frac{-2\pi}{N}id_yf_y}F_a(f_x,f_y).$$

This expression tells us that the Fourier coefficients of the array $b(x,y)$ are identical to the coefficients of the array $a(x,y)$, except for a phase factor. Since taking the absolute values of the Fourier coefficients eliminates any phase factor, it holds that

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

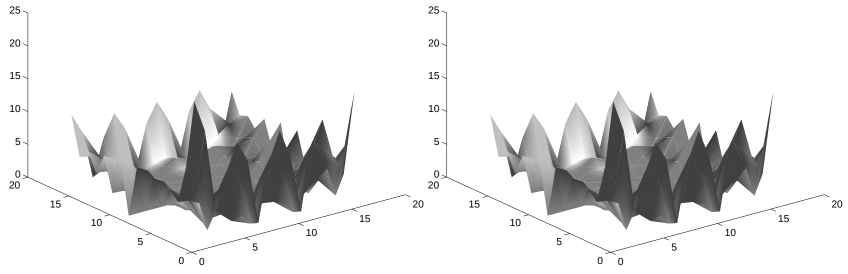


Fig. 12.13. Absolute values of the two-dimensional Fourier transform of the patterns of Figure 12.12

$$\begin{aligned}\|F_b(f_x, f_y)\| &= \|\mathrm{e}^{\frac{-2\pi}{N} i d_x f_x} \mathrm{e}^{\frac{-2\pi}{N} i d_y f_y}\| \|F_a(f_x, f_y)\| \\ &= \|F_a(f_x, f_y)\|,\end{aligned}$$

and this proves that the absolute values of the Fourier coefficients for both patterns are identical.

Figure 12.13 shows the absolute values of the two-dimensional Fourier coefficients for the two patterns of Figure 12.12. They are identical, as expected from the above considerations.

Note that this type of comparison holds as long as the background noise is low. If there is a background with a certain structure it should be displaced together with the patterns, otherwise the analysis performed above does not hold. This is what makes the translation-invariant recognition of patterns a difficult problem when the background is structured. Recognizing faces in real photographs can become extremely difficult when a typical background (a forest, a street, for example) is taken into account.

12.5 Historical and bibliographical remarks

Associative networks have been studied for a long time. Donald Hebb considered in the 1950s how neural assemblies can self-organize into feedback circuits capable of recognizing patterns [308]. Hebbian learning has been interpreted in different ways and several modifications of the basic algorithm have been proposed, but they all have three aspects in common: Hebbian learning is a local, interactive, and time-dependent mechanism [73]. A synaptic phenomenon in the hippocampus, known as long-term potentiation, is thought to be produced by Hebbian modification of the synaptic strength.

There were some other experiments with associative memories in the 1960s, for example the hardware implementations by Karl Steinbuch of his "learning matrix". A preciser mathematical description of associative networks was given by Kohonen in the 1970s [253]. His experiments with many different

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

classes of associative memories contributed enormously to the renewed surge of interest in neural models. Some researchers tried to find biologically plausible models of associative memories following his lead [89]. Some discrete variants of correlation matrices were analyzed in the 1980s, as done for example by Kanerva who considered the case of weight matrices with only two classes of elements, 0 or 1 [233].

In recent times much work has been done on understanding the dynamical properties of recurrent associative networks [231]. It is important to find methods to describe the changes in the basins of attraction of the stored patterns [294]. More recently Haken has proposed his model of a *synergetic computer*, a kind of associative network with a continuous dynamics in which synergetic effects play the crucial role [178]. The fundamental difference from conventional models is the continuous, instead of discrete, dynamics of the network, regulated by some differential equations.

The Moore–Penrose pseudoinverse has become an essential instrument in linear algebra and statistics [14]. It has a simple geometric interpretation and can be computed using standard linear algebraic methods. It makes it possible to deal efficiently with correlated data of the kind found in real applications.

Exercises

1. The Hamming distance between two n-dimensional binary vectors $\mathbf{x}$ and $\mathbf{y}$ is given by $h = \sum_{i=1}^{n}(x_i(1-y_i)+y_i(1-x_i))$. Write h as a function of the scalar product $\mathbf{x}\cdot\mathbf{y}$. Find a similar expression for bipolar vectors. Show that measuring the similarity of vectors using the Hamming distance is equivalent to measuring it using the scalar product.
2. Implement an associative memory capable of recognizing the ten digits from 0 to 9, even in the case of a noisy input.
3. Preprocess the data, so that the pattern recognition process is made translation invariant.
4. Find the inverse of an invertible matrix using the backpropagation algorithm.
5. Show that the pseudoinverse of a matrix is unique.
6. Does the backpropagation algorithm always find the pseudoinverse of any given matrix?
7. Show that the two-dimensional Fourier transform is a composition of two one-dimensional Fourier transforms.
8. Express the two-dimensional Fourier transform as the product of three matrices (see the expression for the one-dimensional FT in Chap. 9).

R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

13

The Hopfield Model

One of the milestones for the current renaissance in the field of neural networks was the associative model proposed by Hopfield at the beginning of the 1980s. Hopfield's approach illustrates the way theoretical physicists like to think about ensembles of computing units. No synchronization is required, each unit behaving as a kind of elementary system in complex interaction with the rest of the ensemble. An energy function must be introduced to harness the theoretical complexities posed by such an approach. The next two sections deal with the structure of Hopfield networks. We then proceed to show that the model converges to a stable state and that two kinds of learning rules can be used to find appropriate network weights.

13.1 Synchronous and asynchronous networks

A relevant issue for the correct design of recurrent neural networks is the adequate synchronization of the computing elements. In the case of McCulloch-Pitts networks we solved this difficulty by assuming that the activation of each computing element consumes a unit of time. The network is built taking this delay into account and by arranging the elements and their connections in the necessary pattern. When the arrangement becomes too contrived, additional units can be included which serve as delay elements. What happens when this assumption is lifted, that is, when the synchronization of the computing elements is eliminated?

13.1.1 Recursive networks with stochastic dynamics

We discussed the design and operation of associative networks in the previous chapter. The synchronization of the output was achieved by requiring that all computing elements evaluate their inputs and compute their output simultaneously. Under this assumption the operation of the associative memory can be described with simple linear algebraic methods. The excitation of the output units is computed using vector-matrix multiplication and evaluating the sign function at each node.

The methods we have used before to avoid dealing explicitly with the synchronization problem have the disadvantage, from the point of view of both biology and physics, that global information is needed, namely a global time. Whereas in conventional computers synchronization of the digital building blocks is achieved using a clock signal, there is no such global clock in biological systems. In a more biologically oriented simulation, global synchronization should thus be avoided. In this chapter we deal with the problem of identifying the properties of neural networks lacking global synchronization.

Networks in which the computing units are activated at different times and which provide a computation after a variable amount of time are stochastic automata. Networks built from this kind of units behave like *stochastic dynamical systems*.

13.1.2 The bidirectional associative memory

Before we start analyzing asynchronous networks we will examine another kind of synchronous associative model with bidirectional edges. We will arrive at the concept of the *energy function* in a very natural way.

We have already discussed recurrent associative networks in which the output of the network is fed back to the input units using additional feedback connections (Figure 12.3). In this way we designed recurrent dynamical systems and tried to determine their fixpoints. However, there is another way to define a recurrent associative memory made up of two layers which send information recursively between them. The input layer contains units which receive the input to the network and send the result of their computation to the output layer. The output of the first layer is transported by bidirectional edges to the second layer of units, which then return the result of their computation back to the first layer using the same edges. As in the case of associative memory models, we can ask whether the network achieves a stable state in which the information being sent back and forth does not change after a few iterations [258]. Such a network (shown in Figure 13.1) is known as a resonance network or *bidirectional associative memory* (BAM). The activation function of the units is the sign function and information is coded using bipolar values.

The network in Figure 13.1 maps an n-dimensional row vector $\mathbf{x}_0$ to a k-dimensional row vector $\mathbf{y}_0$. We denote the $n \times k$ weight matrix of the network by $\mathbf{W}$ so that the mapping computed in the first step can be written as

$$\mathbf{y}_0 = \mathrm{sgn}(\mathbf{x}_0 \mathbf{W}).$$

In the feedback step $\mathbf{y}_0$ is treated as the input and the new computation is

$$\mathbf{x}_1^{\mathrm{T}} = \mathrm{sgn}(\mathbf{W} \mathbf{y}_0^{\mathrm{T}}).$$

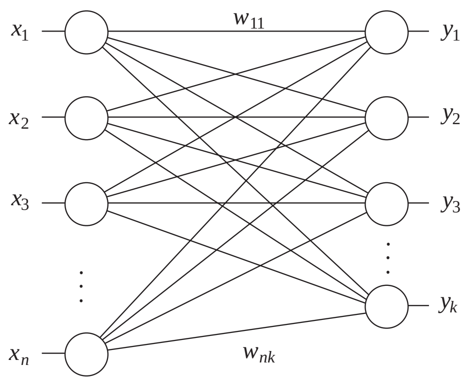


Fig. 13.1. Example of a resonance network (BAM)

A new computation from left to right produces

$$\mathbf{y}_1 = \text{sgn}(\mathbf{x}_1\mathbf{W}).$$

After m iterations the system has computed a set of $m+1$ vector pairs $(\mathbf{x}_0, \mathbf{y}_0), \ldots, (\mathbf{x}_m, \mathbf{y}_m)$ which fulfill the conditions

$$\mathbf{y}_i = \text{sgn}(\mathbf{x}_i\mathbf{W}) \tag{13.1}$$

and

$$\mathbf{x}_{i+1}^{\mathrm{T}} = \text{sgn}(\mathbf{W}\mathbf{y}_i^{\mathrm{T}}). \tag{13.2}$$

The question is whether after some iterations a fixpoint $(\mathbf{x}, \mathbf{y})$ is found. This is the case when both

$$\mathbf{y} = \text{sgn}(\mathbf{x}\mathbf{W}) \quad \text{ and } \quad \mathbf{x}^{\mathrm{T}} = \text{sgn}(\mathbf{W}\mathbf{y}^{\mathrm{T}}) \tag{13.3}$$

hold. The BAM is thus a generalization of a unidirectional associative memory. An input vector, the “key”, can be presented to the network from the left or from the right and, after some iterations, the BAM finds the corresponding complementary vector. As can be seen, no external feedback connections are necessary. The same edges are used for the transmission of information back and forth.

It can be immediately deduced from (13.3) that if a vector pair $(\mathbf{x}, \mathbf{y})$ is given and we want to condition a BAM to accept this pair as a fixed point, Hebbian learning can be used to compute an adequate matrix $\mathbf{W}$. If $\mathbf{W}$ is defined as $\mathbf{W} = \mathbf{x}^{\mathrm{T}}\mathbf{y}$, as prescribed by Hebbian learning, then

$$\mathbf{y} = \text{sgn}(\mathbf{x}\mathbf{W}) = \text{sgn}(\mathbf{x}\mathbf{x}^{\mathrm{T}}\mathbf{y}) = \text{sgn}(\|\mathbf{x}\|^2\mathbf{y}) = \mathbf{y}$$

and also

$$\mathbf{x}^{\mathrm{T}} = \text{sgn}(\mathbf{W}\mathbf{y}^{\mathrm{T}}) = \text{sgn}(\mathbf{x}^{\mathrm{T}}\mathbf{y}\mathbf{y}^{\mathrm{T}}) = \text{sgn}(\mathbf{x}^{\mathrm{T}}\|\mathbf{y}\|^2) = \mathbf{x}^{\mathrm{T}}.$$

If we want to store several vector pairs $(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_m, \mathbf{y}_m)$ in a BAM, then Hebbian learning works better if the vectors $\mathbf{x}_1, \ldots, \mathbf{x}_m$ and $\mathbf{y}_1, \ldots, \mathbf{y}_m$ are pairwise orthogonal within their respective groups, because in that case the perturbation term becomes negligible (refer to Chap. 12).

For a set of m vector pairs the matrix $\mathbf{W}$ is set to

$$\mathbf{W} = \mathbf{x}_1^{\mathrm{T}}\mathbf{y}_1 + \mathbf{x}_2^{\mathrm{T}}\mathbf{y}_2 + \cdots + \mathbf{x}_m^{\mathrm{T}}\mathbf{y}_m.$$

BAMs can be used to build autoassociative networks because the matrices produced by the Hebb rule or by computing the pseudoinverse are symmetric. To see this, define $\mathbf{X}$ as the matrix, each of whose m rows is an n-dimensional vector, so that if $\mathbf{W}$ denotes the connection matrix of an autoassociative memory for those m vectors, then it is true that

$$\mathbf{X} = \mathbf{X}\mathbf{W} \quad \text{and} \quad \mathbf{X}^{\mathrm{T}} = \mathbf{W}\mathbf{X}^{\mathrm{T}},$$

because $\mathbf{W}$ is symmetric. This is just another way of writing the type of computation performed by a BAM.

13.1.3 The energy function

With the BAM we can motivate and explore the concept of an *energy function* in a simple setting. Assume that a BAM is given for which the vector pair $(\mathbf{x}, \mathbf{y})$ is a stable state. If the initial vector presented to the network from the left is $\mathbf{x}_0$, the network will converge to $(\mathbf{x}, \mathbf{y})$ after some iterations. The vector $\mathbf{y}_0$ is computed according to $\mathbf{y}_0 = \mathrm{sgn}(\mathbf{x}_0\mathbf{W})$. If $\mathbf{y}_0$ is now used for a new iteration from the right, excitation of the units in the left layer can be summarized in an excitation vector $\mathbf{e}$ computed according to

$$\mathbf{e}^{\mathrm{T}} = \mathbf{W}\mathbf{y}_0.$$

The vector pair $(\mathbf{x}_0, \mathbf{y}_0)$ is a stable state of the network if $\mathrm{sgn}(\mathbf{e}) = \mathbf{x}_0$. All vectors $\mathbf{e}$ close enough to $\mathbf{x}_0$ fulfill this condition. These vectors differ from $\mathbf{x}_0$ by a small angle and therefore the product $\mathbf{x}_0\mathbf{e}^{\mathrm{T}}$ is larger than for other vectors of the same length but further away from $\mathbf{x}_0$. The product

$$E = -\mathbf{x}_0\mathbf{e}^{\mathrm{T}} = -\mathbf{x}_0\mathbf{W}\mathbf{y}_0^{\mathrm{T}}$$

is therefore smaller (because of the minus sign) if the vector $\mathbf{W}\mathbf{y}_0^{\mathrm{T}}$ lies closer to $\mathbf{x}_0$. The scalar value E can be used as a kind of index of convergence to the stable states of an associative memory. We call E the *energy function* of the network.

Definition 16. *The energy function E of a BAM with weight matrix $\mathbf{W}$, in which the output $\mathbf{y}_i$ of the right layer of units is computed in the i-th iteration according to equation* (13.1) *and the output $\mathbf{x}_i$ of the left layer is computed according to* (13.2) *is given by*

$$E(\mathbf{x}_i, \mathbf{y}_i) = -\frac{1}{2}\mathbf{x}_i\mathbf{W}\mathbf{y}_i^{\mathrm{T}}. \tag{13.4}$$

The factor 1/2 will be useful later and is just a scaling constant for the energy function. In the following sections we show that the energy function assumes locally minimal values at stable states. The energy function can also be generalized to arbitrary vectors $\mathbf{x}$ and $\mathbf{y}$.

Up to this point we have only considered units with the sign function as activation nonlinearity in the type of associative memories we have discussed. If we now consider units with a threshold and the step function as its activation function, we must use a more general expression for the energy function. This can be done by extending the input vectors with an additional constant component. Each n-dimensional vector $\mathbf{x}$ will be transformed into the vector $(x_1, \ldots, x_n, 1)$. We proceed in a similar way with the k-dimensional vector $\mathbf{y}$. The weight matrix $\mathbf{W}$ must be extended to a new matrix $\mathbf{W}'$ with an additional row and column. The negative thresholds of the units in the right layer of the BAM are included in row $n+1$ of $\mathbf{W}'$, whereas the negative thresholds of the units in the left are used as the entries of the column $k+1$ of the weight matrix. The entry $(n+1, k+1)$ of the weight matrix can be set to zero. This transformation is equivalent to the introduction of an additional unit with constant output 1 into each layer. The weight of each edge from a constant unit to each one of the others is the negative threshold of the connected unit. It is straightforward to deduce that the energy function of the extended network can be written as

$$E(\mathbf{x}_i, \mathbf{y}_i) = -\frac{1}{2}\mathbf{x}_i \mathbf{W} \mathbf{y}_i^{\mathrm{T}} + \frac{1}{2}\theta_r \mathbf{y}_i^{\mathrm{T}} + \frac{1}{2}\mathbf{x}_i \theta_\ell^{\mathrm{T}}. \tag{13.5}$$

The row vector of thresholds of the k units in the left layer is denoted in the above expression by θ_ℓ. The row vector of thresholds of the n units in the right layer is denoted by θ_r.

13.2 Definition of Hopfield networks

So far we have considered only conventional or bidirectional associative memories working with synchronized units. Dropping the assumption of simultaneous firing of the computing elements leads to the appearance of novel network properties.

13.2.1 Asynchronous networks

In an asynchronous network each unit computes its excitation at random times and changes its state to 1 or -1 independently of the others and according to the sign of its total excitation. The probability of two units firing simultaneously is zero. Consequently, the same dynamics can be obtained by selecting one unit randomly, computing its excitation and updating its state accordingly. There will not be any delay between computation of the excitation and state update. We adopt the additional simplification that the state of a unit is not changed if the total excitation is zero. This means that we leave the sign function undefined for the argument zero. Asynchronous networks are of course more realistic models of biological networks, although the assumption of zero delay in the computation and transmission of signals lacks any biological basis.

Using the energy function it can be shown that a BAM arrives at a stable state after a finite number of iterations. A stable state is a vector pair $(\mathbf{x}, \mathbf{y})$ which fulfills the conditions (13.3). When a BAM reaches this state pair, no component of the bipolar vectors $\mathbf{x}$ and $\mathbf{y}$ can be changed without contradicting (13.3). The vector pair $(\mathbf{x}, \mathbf{y})$ is therefore also a stable state for an asynchronous network.

Proposition 19. *A bidirectional associative memory with an arbitrary weight matrix* $\mathbf{W}$ *reaches a stable state in a finite number of iterations using either synchronous or asynchronous updates.*

Proof. For a vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, a vector $\mathbf{y} = (y_1, y_2, \ldots, y_k)$ and an $n \times k$ weight matrix $\mathbf{W} = \{w_{ij}\}$ the energy function is the bilinear form

$$E(\mathbf{x}, \mathbf{y}) = -\frac{1}{2}(x_1, x_2, \ldots, x_n) \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1k} \\ w_{21} & w_{22} & \cdots & w_{2k} \\ \vdots & & \ddots & \vdots \\ w_{n1} & w_{n2} & \cdots & w_{nk} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{pmatrix}.$$

The value of $E(\mathbf{x}, \mathbf{y})$ can be computed by multiplying first $\mathbf{W}$ by $\mathbf{y}^{\mathrm{T}}$ and the result with $-\mathbf{x}/2$. The product of the i-th row of $\mathbf{W}$ and $\mathbf{y}^{\mathrm{T}}$ represents the excitation of the i-th unit in the left layer. If we denote these excitations by $g_1, g_2, \ldots, g_n$ the above expression transforms to

$$E(\mathbf{x}, \mathbf{y}) = -\frac{1}{2}(x_1, x_2, \ldots, x_n) \begin{pmatrix} g_1 \\ g_2 \\ \vdots \\ g_n \end{pmatrix}.$$

We can also compute $E(\mathbf{x}, \mathbf{y})$ multiplying first $\mathbf{x}$ by $\mathbf{W}$. The product of the i-th column of $\mathbf{W}$ with $\mathbf{x}$ corresponds to the excitation of unit i in the right layer. If we denote these excitations by $e_1, e_2, \ldots, e_k$, the expression for $E(\mathbf{x}, \mathbf{y})$ can be written as

$$E(\mathbf{x}, \mathbf{y}) = -\frac{1}{2}(e_1, e_2, \ldots, e_k) \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{pmatrix}.$$

Therefore, the energy function can be written in the two equivalent forms

$$E(\mathbf{x}, \mathbf{y}) = -\frac{1}{2}\sum_{i=1}^{k} e_i y_i \quad \text{and} \quad E(\mathbf{x}, \mathbf{y}) = -\frac{1}{2}\sum_{i=1}^{n} g_i x_i.$$

In asynchronous networks at each time t we randomly select a unit from the left or right layer. The excitation is computed and its sign is the new activation of the unit. If the previous activation of the unit remains the same after this operation, then the energy of the network has not changed.

The state of unit i on the left layer will change only when the excitation g_i has a different sign than x_i, the present state. The state is updated from x_i to x'_i, where x'_i now has the same sign as g_i. Since the other units do not change their state, the difference between the previous energy $E(\mathbf{x},\mathbf{y})$ and the new energy $E(\mathbf{x}',\mathbf{y})$ is

$$E(\mathbf{x},\mathbf{y}) - E(\mathbf{x}',\mathbf{y}) = -\frac{1}{2} g_i (x_i - x'_i).$$

Since both x_i and $-x_i$ have a different sign than g_i it follows that

$$E(\mathbf{x},\mathbf{y}) - E(\mathbf{x}',\mathbf{y}) > 0.$$

The new state $(\mathbf{x}',\mathbf{y})$ has a lower energy than the original state $(\mathbf{x},\mathbf{y})$. The same argument can be made if a unit on the right layer has been selected, so that for the new state $(\mathbf{x},\mathbf{y}')$ it holds that

$$E(\mathbf{x},\mathbf{y}) - E(\mathbf{x},\mathbf{y}') > 0,$$

whenever the state of a unit in the right layer has been flipped.

Any update of the network state reduces the total energy. Since there are only a finite number of possible combinations of bipolar states, the process must stop at some point, that is, a state $(\mathbf{a},\mathbf{b})$ is found whose energy cannot be further reduced. The network has fallen into a local minimum of the energy function and the state $(\mathbf{a},\mathbf{b})$ is an attractor of the system. □

The above proposition also holds for synchronous networks, since these can be considered as a special case of asynchronous dynamics. Note that the proposition puts conditions on the matrix $\mathbf{W}$. This means that any given real matrix $\mathbf{W}$ possesses bidirectional stable bipolar states.

13.2.2 Examples of the model

In 1982 the American physicist John Hopfield proposed an asynchronous neural network model which made an immediate impact in the AI community. It is a special case of a bidirectional associative memory, but chronologically it was proposed before the BAM.

In the Hopfield model it is assumed that the individual units preserve their individual states until they are selected for a new update. The selection is made randomly. A Hopfield network consists of n totally coupled units, that is, each unit is connected to all other units except itself. The network is symmetric because the weight w_{ij} for the connection between unit i and unit j is equal to the weight w_{ji} of the connection from unit j to unit i. This can be interpreted as meaning that there is a single bidirectional connection between both units. The absence of a connection from each unit to itself avoids a permanent feedback of its own state value [198].

Figure 13.2 shows an example of a network with three units. Each one of them can assume the state 1 or -1. A Hopfield network can also be interpreted as an asynchronous BAM in which the left and right layers of units have fused to a single layer. The connections in a Hopfield network with n units can be represented using an $n \times n$ weight matrix $\mathbf{W} = \{w_{ij}\}$ with a zero diagonal.

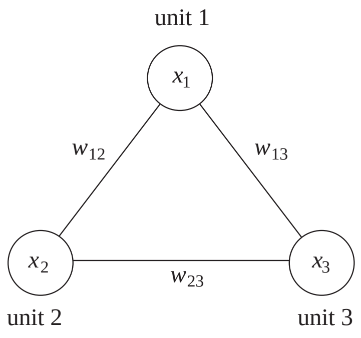


Fig. 13.2. A Hopfield network of three units

It is easy to show that if the weight matrix does not contain a zero diagonal, the network dynamics does not necessarily lead to stable states. The weight matrix

$$\mathbf{W} = \begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix},$$

for example, transforms the state vector $(1, 1, 1)$ into the state vector $(-1, -1, -1)$ and conversely. In the case of asynchronous updating, the network chooses randomly among the eight possible network states.

A connection matrix with a zero diagonal can also lead to oscillations in the case where the weight matrix is not symmetric. The weight matrix

$$\mathbf{W} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$$

describes the network of Figure 13.3. It transforms the state vector $(1, -1)$ into the state vector $(1, 1)$ when the network is running asynchronously. After this transition the state $(-1, 1)$ can be updated to $(-1, -1)$ and finally to $(1, -1)$. The state vector changes cyclically and does not converge to a stable state.

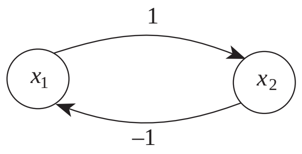


Fig. 13.3. Network with asymmetric connections

The symmetry of the weight matrix and a zero diagonal are thus *necessary conditions* for the convergence of an asynchronous totally connected network to a stable state. These conditions are also sufficient, as we show later.

The units of a Hopfield network can be assigned a threshold θ different from zero. In this case each unit selected for a state update adopts the state 1 if its total excitation is greater than θ, otherwise the state -1. This is the activation rule for perceptrons, so that we can think of Hopfield networks as asynchronous recurrent networks of perceptrons.

The energy function of a Hopfield network composed of units with thresholds different from zero can be defined in a similar way as for the BAM. In this case the vector $\mathbf{y}$ of equation (13.5) is $\mathbf{x}$ and we let $\theta = \theta_\ell = \theta_r$.

Definition 17. *Let* $\mathbf{W}$ *denote the weight matrix of a Hopfield network of* n *units and let* θ *be the* n*-dimensional row vector of units' thresholds. The energy* $E(\mathbf{x})$ *of a state* $\mathbf{x}$ *of the network is given by*

$$E(\mathbf{x}) = -\frac{1}{2}\mathbf{x}\mathbf{W}\mathbf{x}^{\mathrm{T}} + \theta\mathbf{x}^{\mathrm{T}}.$$

The energy function can also be written in the form

$$E(\mathbf{x}) = -\frac{1}{2}\sum_{j=1}^{n}\sum_{i=1}^{n} w_{ij}x_i x_j + \sum_{i=1}^{n}\theta_i x_i.$$

The factor 1/2 is used because the identical terms $w_{ij}x_ix_j$ and $w_{ji}x_jx_i$ are present in the double sum.

The energy function of a Hopfield network is a quadratic form. A Hopfield network always finds a local minimum of the energy function. It is thus interesting to look at an example of the shape of such an energy function. Figure 13.4 shows a network of just two units with threshold zero. It is obvious that the only stable states are $(1, -1)$ and $(-1, 1)$. In any other state, one of the units forces the other to change its state to stabilize the network. Such a network is a flip-flop, a logic component with two outputs which assume complementary logic values.

The energy function of a flip-flop with weights $w_{12} = w_{21} = -1$ and two units with threshold zero is given by

$$E(x_1, x_2) = x_1 x_2,$$

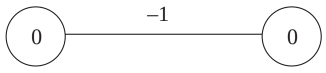


Fig. 13.4. A flip-flop

where x_1 and x_2 denote the states of the first and second units respectively. Figure 13.5 shows the energy function for the so-called continuous Hopfield model [199] in which the unit's states can assume all real values between 0 and 1. In the network of Figure 13.4 only the four discrete states $(1, 1)$, $(1, -1)$, $(-1, 1)$ and $(-1, -1)$ are allowed. The energy function has local minima at $(1, -1)$ and $(-1, 1)$. A flip-flop can therefore be interpreted as a network capable of storing one of the states $(1, -1)$ or $(-1, 1)$.

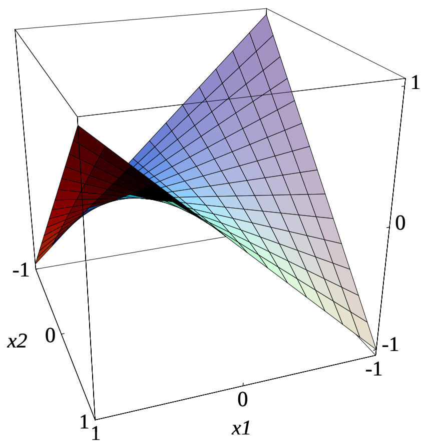


Fig. 13.5. Energy function of a flip-flop

Hopfield networks can also be used to compute logical functions. Conjunction, for example, can be implemented with a network of three units. The states of two units are set and remain fixed during the computation (clamping their states). Only the third unit can change its state. If the network weights and the unit thresholds have the appropriate values, the unconstrained unit will assume a state which corresponds to the conjunction of the two clamped states.

Figure 13.6 shows a network for the computation of the logical disjunction of two Boolean values x_1 and x_2. The input is clamped and after some time the network settles to a state which corresponds to the disjunction of x_1 and x_2. The constants "true" and "false" correspond to the numerical values 1 and -1. In this network the thresholds of the clamped units and their mutual connections play no role in the computation.

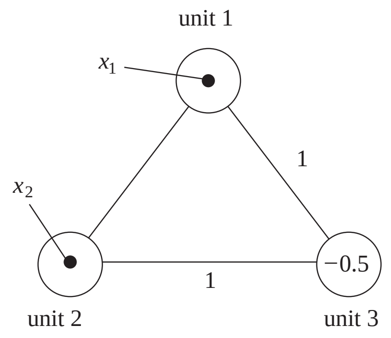


Fig. 13.6. Network for the computation of the OR function

Since the individual units of the network are perceptrons, the question of whether there are logic functions which cannot be computed by a Hopfield network of a given size arises. This is the case in our next example. Assume that a Hopfield network of three units should store the set of stable states given by the following table:

unit	1	2	3
state 1	-1	-1	-1
state 2	1	-1	1
state 3	-1	1	1
state 4	1	1	-1

From the point of view of the third unit (third column) this is the XOR function. If the four vectors shown above are to become stable states of the network, the third unit cannot change state when any of these four vectors has been loaded in the network. In this case the third unit should be capable of linearly separating the vectors $(-1,-1)$ and $(1,1)$ from the vectors $(-1,1)$ and $(1,-1)$, which we know is impossible. The same argument is valid for any of the three units, since the table given above remains unchanged after a permutation of the units' labels. This shows that no Hopfield network of three units can have these stable states. However, the XOR problem can be solved if the network is extended to four units. The network of Figure 13.7 can assume the following stable states, if adequate weights and thresholds are selected:

unit	1	2	3	4
state 1	-1	-1	-1	1
state 2	1	-1	1	1
state 3	-1	1	1	1
state 4	1	1	-1	-1

The third column represents the XOR function of the two first columns. The fourth column corresponds to an auxiliary unit, whose state can be set from outside. The unknown weights can be found using the learning algorithms described in the next sections.

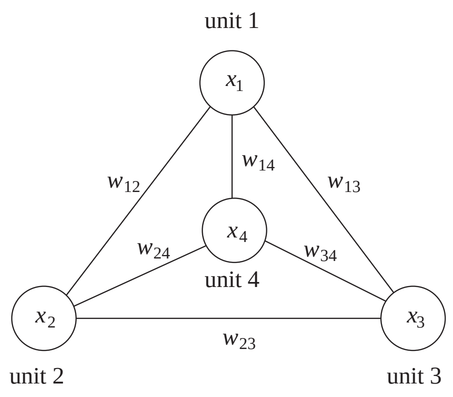


Fig. 13.7. Network for the computation of XOR

13.2.3 Isomorphism between the Hopfield and Ising models

Physicists have analyzed the Hopfield model in such exquisite detail because it is isomorphic to the *Ising model* of magnetism (at temperature zero) [25]. Ising proposed the model which now bears his name more than 70 years ago in order to describe some properties of ensembles of elementary magnets [214].

In general, the Ising model can be used to describe those systems made of particles capable of adopting one of two states. In the case of ferromagnetic materials, their atoms can be modeled as particles of spin 1/2 (*up*) or spin −1/2 (*down*). The spin points in the direction of the magnetic field. All tiny magnets interact with each other. This causes some of the atoms to flip their spin until equilibrium is reached and the total magnetization of the material reaches a constant level, which is the sum of the individual spins. With these few assumptions we can show that the energy function deduced from the Ising model has the same form as the energy function of Hopfield networks.

The total magnetic field h_i sensed by the atom i in an ensemble of particles is the sum of the fields induced by each atom and the external field h^* (if present), that is

$$h_i = \sum_{j=1}^{n} w_{ij} x_j + h^*, \tag{13.6}$$

where w_{ij} represents the magnitude of the magnetic coupling between the atoms labeled i and j. The magnetic coupling changes according to the distance between atoms and the magnetic permeability of the environment. The

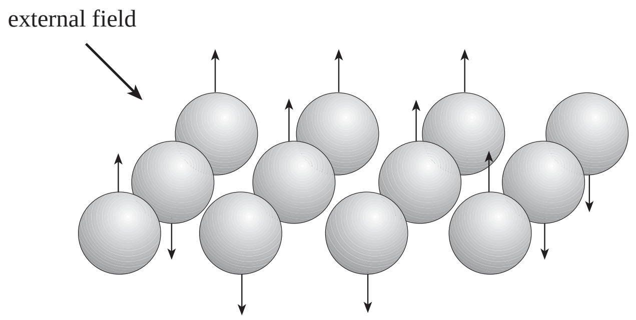


Fig. 13.8. Particles with two possible spins

potential energy E of a certain state $(x_1, x_2, \ldots, x_n)$ of an Ising material can be derived from (13.6) and has the form

$$E = -\frac{1}{2} \sum_{i,j}^{n} w_{ij} x_i x_j + \sum_{i}^{n} -h^* x_i. \tag{13.7}$$

In paramagnetic materials the coupling constants are zero. In ferromagnetic materials the constants w_{ij} are all positive, which leads in turn to a significant coupling of the spin states.

Equation (13.7) is isomorphic to the energy function of Hopfield networks. This is why the term *energy function* is used in the first place. Both systems are dynamically equivalent, but only in the case of zero temperature, since the system behaves deterministically at each state update. Later on, when we consider *Boltzmann machines*, we will accept a time-varying temperature and stochastic state updates as in the full Ising model.

13.3 Converge to stable states

It is easy to show that Hopfield models always converge to stable states. The proof of this fact relies on analysis of the new value of the energy function after each state update.

13.3.1 Dynamics of Hopfield networks

Before going into the details of the convergence proof, we analyze two simple examples and compute the energy levels of all their possible states. Figure 13.9 shows a network composed of three units with arbitrarily chosen weights and thresholds. The network can adopt any of eight possible states whose transitions we want to visualize. Figure 13.10 shows a diagram of all possible state transitions for the network of Figure 13.9. The vertical axis represents the energy of the network defined in the usual way. Each state of the network is represented by an oval located at its precise *energy level*. The arrows show the state transitions allowed. Each transition has the same probability because the probability of selecting one of the three units for a state transition is uniform and equal to 1/3. Note that the diagram does not show the few transitions in which a state returns to itself.

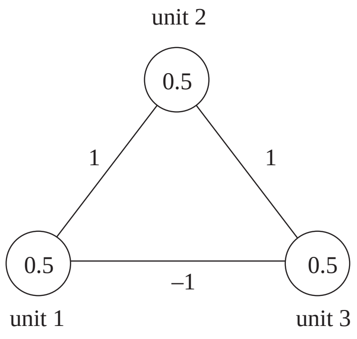


Fig. 13.9. Example of a Hopfield network

We can make other interesting observations in the transition diagram. The state $(1, -1, 1)$, for example, is extremely unstable. The probability of leaving it at the next iteration is 1, because three different transitions to other states are possible, each with probability 1/3. The state $(-1, 1, 1)$ is relatively stable because the probability of leaving it at the next iteration is just 1/3. There is only a single stable state, namely the vector $(-1, -1, -1)$, as the reader can readily verify. The only two states without a predecessor are shown in gray. In the theory of cellular automata, such “urstates” are called *garden of Eden* configurations. They cannot be arrived at, they can only be induced from the outside before the automaton starts working.

The network in Figure 13.11 has the same structure as the network considered previously, but the weights and thresholds have the opposite sign. The diagram of state transitions (Figure 13.12) is the inversion of the diagram in Figure 13.10. The new network has two stable states and just one state without predecessors. As can be seen from the diagrams, the dynamic of the Hopfield model is always the same: the energy of the system eventually reaches a local minimum and the state of the network can no longer change.

13.3.2 Convergence proof

We can now proceed to prove that, in general, Hopfield models behave in the way shown in the last two examples.

Proposition 20. *A Hopfield network with n units and asynchronous dynamics, which starts from any given network state, eventually reaches a stable state at a local minimum of the energy function.*

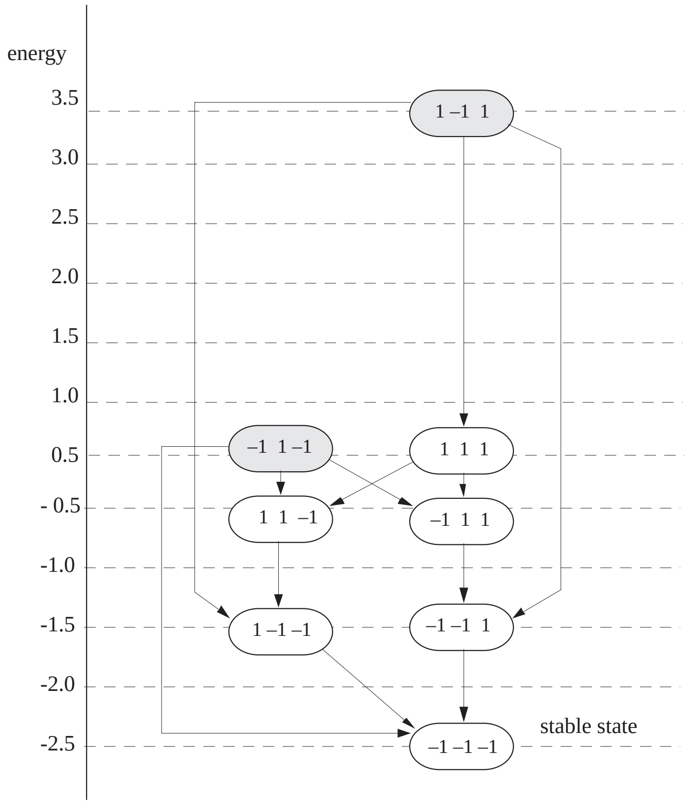


Fig. 13.10. State transitions for the network of Figure 13.9

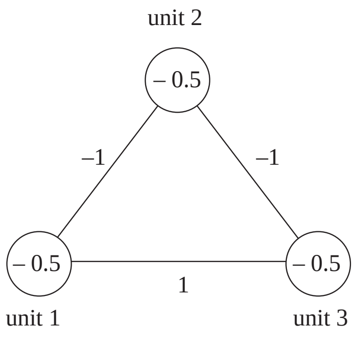


Fig. 13.11. Second example of a Hopfield network

Proof. The energy function of a state $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ of a Hopfield network with n units is given by

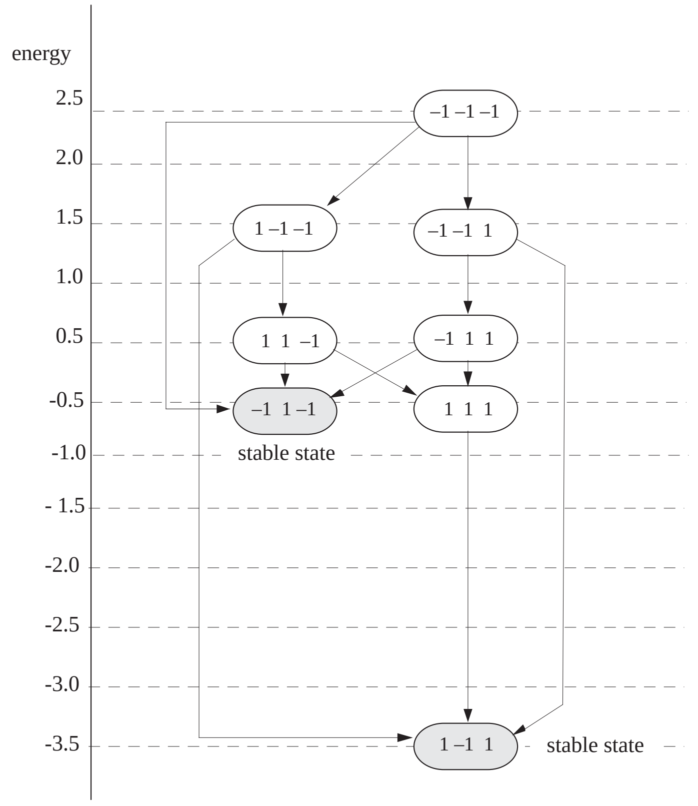


Fig. 13.12. State transitions for the network of Figure 13.11

$$E(\mathbf{x}) = -\frac{1}{2}\sum_{j=1}^{n}\sum_{i=1}^{n} w_{ij}x_ix_j + \sum_{i=1}^{n}\theta_i x_i, \tag{13.8}$$

where the terms involved are defined as usual. If during the current iteration unit k is selected and does not change its state, then the energy of the system does not change either. If the state of the unit is changed in the update operation, the network reaches a new global state $\mathbf{x}' = (x_1, \ldots, x'_k, \ldots, x_n)$ for which the new energy is $E(\mathbf{x}')$. The difference between $E(\mathbf{x})$ and $E(\mathbf{x}')$ is given by all terms in the summation in (13.8) which contain x_k and x'_k, that is

$$E(\mathbf{x}) - E(\mathbf{x}') = (-\sum_{j=1}^{n} w_{kj}x_kx_j + \theta_k x_k) - (-\sum_{j=1}^{n} w_{kj}x'_kx_j + \theta_k x'_k).$$

The factor 1/2 disappears from the computation because the terms $w_{kj}x_kx_j$ appear twice in the double sum of (13.8). Since $w_{kk} = 0$ we can rewrite the above equation as

$$
\begin{aligned}
E(\mathbf{x}) - E(\mathbf{x}') &= -(x_k - x'_k)\sum_{j=1}^{n} w_{kj}x_j + \theta_k(x_k - x'_k) \\
&= -(x_k - x'_k)(\sum_{j=1}^{n} w_{kj}x_j - \theta_k),
\end{aligned}
$$

from which we finally obtain

$$E(\mathbf{x}) - E(\mathbf{x}') = -(x_k - x'_k)e_k,$$

where e_k denotes the total excitation of unit k (including subtraction of the threshold). The excitation e_k has a different sign from x_k and $-x'_k$, because otherwise the unit state would not have been changed. This means that the product $-(x_k - x'_k)e_k$ is positive and therefore

$$E(\mathbf{x}) - E(\mathbf{x}') > 0.$$

This shows that every time the state of a unit is altered, the total energy of the network is reduced. Since there is only a finite set of possible states, the network must eventually reach a state for which the energy cannot be reduced further. It is a stable state of the network, as we wanted to prove. □

There is a simpler proof of the last proposition, which has the advantage of offering a nice visualization of the dynamics of a Hopfield network [74]. Assume that we classify the units of a network according to their state: the first set contains the units with state 1, the second set the units with state −1. There are edges linking every unit with all the others, so that some edges go from one set to the other. We now randomly select one of the units and compute its "attraction" by the units in its own set and the attraction by the units in the other set. The "attraction" is the sum of the weights of all edges between a unit and the units in its set or in the other one. If the attraction from the outside is greater than the attraction from its own set, the unit changes sides by altering its state. If the external attraction is lower than the internal, the unit keeps its current state. This procedure is repeated several times, each time selecting one of the units randomly. It corresponds to the updating strategy of a Hopfield network. Figure 13.13 shows an example in which the attraction from the outside is greater than the internal one. The selected unit must change sides. It is clear that the network must eventually reach a stable state, because the sum of the weights of all edges connecting one set to the other can only become lower in the course of time. Since the number of possible network states is finite, a global state must be reached in which the attraction of one set by the other cannot be further reduced. This is the task known in combinatorics as the *minimal cut* problem, in which we want to find a cut of minimal flow in a graph. The procedure described always finds a locally minimal cut.

The wording of Proposition 20 has been carefully chosen. That the network "eventually" settles in a stable state, means that the probability of not

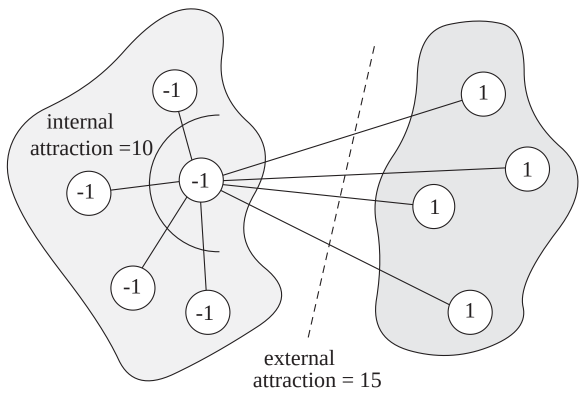


Fig. 13.13. Attraction from the inside and from the outside of a unit's class

reaching such a state approaches zero as the number of iterations increases. It would be possible to select always one and the same unit for computation of the excitation, and in this case the network would stay in deadlock. Since the units are selected randomly, the probability of such pathological behavior falls to zero as time progresses.

In the proof of Proposition 20 only the symmetry and the zero diagonal of the weight matrix were used. The proof of convergence is very similar to the proof of convergence for the BAM. However, in the case of a BAM the decisive property was the independence of a unit's state from its own excitation. This is also the case for Hopfield networks, since no unit feeds its own state back into itself, i.e., the diagonal of the weight matrix is zero.

13.3.3 Hebbian learning

A Hopfield network can be used as an associative memory. If we want to "imprint" m different stable states in the network we have to find adequate weights for the connections. In the case of the BAM we already mentioned that Hebbian learning is a possible alternative. Since Hopfield networks are a specialization of BAM networks, we also expect Hebbian learning to be applicable in this case. Let us first discuss the case of a Hopfield network with n units and threshold zero.

Hebbian learning is implemented by loading the m selected n-dimensional stable states $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m$ on the network and by updating the network's weights (initially set to zero) after each presentation according to the rule

$$w_{ij} \leftarrow w_{ij} + x_i^k x_j^k, \quad i, j = 1, \ldots, n \quad \text{and} \quad i \neq j.$$

The symbols x_i^k and x_j^k denote the i-th and j-th component respectively of the vector $\mathbf{x}_k$. The only difference from an autoassociative memory is the requirement of a zero diagonal. After presentation of the first vector $\mathbf{x}_1$ the weight matrix is given by the expression

$$\mathbf{W}_1 = \mathbf{x}_1^{\mathrm{T}}\mathbf{x}_1 - \mathbf{I},$$

where $\mathbf{I}$ denotes the $n \times n$ identity matrix. Subtraction of the identity matrix guarantees that the diagonal of $\mathbf{W}$ becomes zero, since for any bipolar vector $\mathbf{x}_i$ it holds that $x_k^i x_k^i = 1$. Obviously $\mathbf{W}_1$ is a symmetric matrix.

The minimum of the energy function of a Hopfield network with the weight matrix $\mathbf{W}_1$ is located at $\mathbf{x}_1$ because

$$E(\mathbf{x}) = -\frac{1}{2}\mathbf{x}\mathbf{W}_1\mathbf{x}^{\mathrm{T}} = -\frac{1}{2}(\mathbf{x}\mathbf{x}_1^{\mathrm{T}}\mathbf{x}_1\mathbf{x}^{\mathrm{T}} - \mathbf{x}\mathbf{x}^{\mathrm{T}})$$

and $\mathbf{x}\mathbf{x}^{\mathrm{T}} = n$ for bipolar vectors. This means that the function

$$E(\mathbf{x}) = -\frac{1}{2}\|\mathbf{x}\mathbf{x}_1^{\mathrm{T}}\|^2 + \frac{n}{2}$$

has a local minimum at $\mathbf{x} = \mathbf{x}_1$. In this case it holds that

$$E(\mathbf{x}) = -\frac{n^2}{2} + \frac{n}{2}.$$

This shows that $\mathbf{x}_1$ is a stable state of the network.

In the case of m different vectors $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m$ the matrix $\mathbf{W}$ is defined as

$$\mathbf{W} = (\mathbf{x}_1\mathbf{x}_1^{\mathrm{T}} - \mathbf{I}) + (\mathbf{x}_2^{\mathrm{T}}\mathbf{x}_2 - \mathbf{I}) + \cdots + (\mathbf{x}_m^{\mathrm{T}}\mathbf{x}_m - \mathbf{I}),$$

or equivalently

$$\mathbf{W} = \mathbf{x}_1^{\mathrm{T}}\mathbf{x}_1 + \mathbf{x}_2^{\mathrm{T}}\mathbf{x}_2 + \cdots + \mathbf{x}_m^{\mathrm{T}}\mathbf{x}_m - m\mathbf{I}.$$

If the network is initialized with the state $\mathbf{x}_1$, the vector $\mathbf{e}$ of the excitation of the units is

$$\begin{aligned}\mathbf{e} &= \mathbf{x}_1\mathbf{W}\\ &= \mathbf{x}_1\mathbf{x}_1^{\mathrm{T}}\mathbf{x}_1 + \mathbf{x}_1\mathbf{x}_2^{\mathrm{T}}\mathbf{x}_2 + \cdots + \mathbf{x}_1\mathbf{x}_m^{\mathrm{T}}\mathbf{x}_m - m\mathbf{x}_1\mathbf{I}\\ &= (n-m)\mathbf{x}_1 + \sum_{j=2}^{m}\alpha_{1j}\mathbf{x}_j.\end{aligned}$$

The constants $\alpha_{12}, \alpha_{13}, \ldots, \alpha_{1m}$ represent the scalar products of the first vector with each one of the other $m-1$ vectors $\mathbf{x}_2, \ldots, \mathbf{x}_m$. The state $\mathbf{x}_1$ is stable when $m < n$ and the perturbation term $\sum_{j=2}^{m}\alpha_{1j}\mathbf{x}_j$ is small. In this case it holds that

$$\mathrm{sgn}(\mathbf{e}) = \mathrm{sgn}(\mathbf{x}_1)$$

as desired. The same argumentation can be used for any of the other vectors. The best results are achieved with Hebbian learning when the vectors $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m$ are orthogonal or close to orthogonal, just as in the case of any other associative memory.

13.4 Equivalence of Hopfield and perceptron learning

Hebbian learning is a simple rule which is useful for the computation of the weight matrix in Hopfield networks. However, sometimes Hebbian learning cannot find a weight matrix for which m given vectors are stable states, although such a matrix exists. If the vectors to be stored lie near each other, the perturbation term can grow so large as to preclude a solution by Hebbian learning. In this case another learning rule is needed, which is a variant of perceptron learning.

13.4.1 Perceptron learning in Hopfield networks

Let us consider Hopfield networks composed of units with a non-zero threshold and the step function as activation function. The units adopt state 1 when the excitation is greater than the threshold and otherwise the state -1. The units are just perceptrons and it is straightforward to assume that perceptron learning could be used for determination of the weights and thresholds of the network for a given learning problem.

Let n denote the number of units in a Hopfield network, let $\mathbf{W} = \{w_{ij}\}$ be the $n \times n$ weight matrix, and let θ_i denote the threshold of unit i. If a vector $\mathbf{x} = (x_1, \ldots, x_n)$ is given to be "imprinted" on the network, this vector will be a stable state only when, if loaded in the network, the network global state does not change. This is the case if for every unit its excitation minus its threshold has the same sign as the current state (the value zero is assigned the minus sign). This means that the following n inequalities must hold:

$$\begin{array}{lccccc}
\text{For unit 1}: \operatorname{sgn}(x_1)(0 & + x_2 w_{12} & + x_3 w_{13} & + & \cdots & \\
 & + x_n w_{1n} & - \ \ \theta_1) & < & 0 & \\
\text{For unit 2}: \operatorname{sgn}(x_2)(x_1 w_{21} & + \ \ 0 & + x_3 w_{23} & + & \cdots & \\
 & + x_n w_{2n} & - \ \ \theta_2) & < & 0 & \\
\vdots & & & & & \\
\text{For unit n}: \operatorname{sgn}(x_n)(x_1 w_{n1} & + x_2 w_{n2} & + & \cdots & + x_{n-1} w_{nn-1} & \\
 & + \ \ 0 & - \ \ \theta_n) & < & 0 &
\end{array}$$

The factor $\operatorname{sgn}(x_i)$ is used in each inequality to obtain always the same inequality operator ("less than"). Only the $n(n-1)/2$ non-zero entries of the weight matrix as well as the n thresholds of the units appear in these inequalities. Let $\mathbf{v}$ denote a vector of dimension $n + n(n-1)/2$ whose components are the non-diagonal entries w_{ij} of the weight matrix $\mathbf{W}$ (with $i < j$ so as to consider each weight only once) and the n thresholds with minus sign. The vector $\mathbf{v}$ is given by

$$\mathbf{v} = (\underbrace{w_{12}, w_{13}, \ldots, w_{1n}}_{n-1}, \underbrace{w_{23}, w_{24}, \ldots, w_{2n}}_{n-2}, \ldots, \underbrace{w_{n-1n}}_{1}, \underbrace{-\theta_1, \ldots, -\theta_n}_{n}).$$

The vector $\mathbf{x}$ is transformed into n auxiliary vectors $\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_n$ of dimension $n + n(n-1)/2$ given by the expression

$$\mathbf{z}_1 = (\underbrace{x_2, x_3, \ldots, x_n}_{n-1}, 0, 0, \ldots, \underbrace{1, 0, \ldots, 0}_{n})$$
$$\mathbf{z}_2 = (\underbrace{x_1, 0, \ldots, 0}_{n-1}, \underbrace{x_3, \ldots, x_n}_{n-2}, 0, 0, \ldots, \underbrace{0, 1, \ldots, 0}_{n})$$
$$\vdots$$
$$\mathbf{z}_n = (\underbrace{0, 0, \ldots, x_1}_{n-1}, \underbrace{0, 0, \ldots, x_2}_{n-2}, 0, 0, \ldots, \underbrace{0, 0, \ldots, 1}_{n}).$$

The components of the vectors $\mathbf{z}_1, \ldots, \mathbf{z}_n$ were defined so that the previous inequalities for each unit can be written in the equivalent form

$$\begin{array}{l} \text{unit } 1 \ \operatorname{sgn}(x_1)\mathbf{z}_1 \cdot \mathbf{v} > 0 \\ \text{unit } 2 \ \operatorname{sgn}(x_2)\mathbf{z}_2 \cdot \mathbf{v} > 0 \\ \vdots \\ \text{unit } n \ \operatorname{sgn}(x_n)\mathbf{z}_n \cdot \mathbf{v} > 0 \end{array}$$

The vectors $\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_n$ can always be defined in this way. We will not write down the exact transformation rule here because it is rather involved.

The last set of inequalities shows that the solution to the original problem is found by computing a linear separation of the vectors $\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_n$. The vectors which belong to the positive half-space are those for which $\operatorname{sgn}(x_i)$ holds. The vectors which belong to the negative half-space are those for which $\operatorname{sgn}(x_i) = -1$. This problem can be solved using perceptron learning, which allows us to compute the vector $\mathbf{v}$ of weights needed for the linear separation, and from this we can deduce the weight matrix $\mathbf{W}$.

In the case where m vectors $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m$ are given to be imprinted in the Hopfield network, we have to use the above transformation for every one of them. Each vector is transformed into n auxiliary vectors, so that at the end we have nm different auxiliary vectors which must be linearly separated. If they are actually linearly separable, perceptron learning will find the solution to the problem, coded in the vector $\mathbf{v}$ of the transformed perceptron.

The analysis performed above shows that it is possible to transform a learning problem in a Hopfield network with n units into a learning problem for a perceptron of dimension $n + n(n-1)/2$, that is, $n(n+1)/2$. Figure 13.14 shows an example of a Hopfield network that can be transformed into the equivalent perceptron to the right. The three-dimensional Hopfield problem is transformed in this way into a learning problem for a six-dimensional perceptron.

Each iteration of the perceptron learning algorithm updates only the weights of the edges attached to a single unit and its threshold. For example, if a correction is needed because of the sign of $\mathbf{z}_1 \cdot \mathbf{v}$, then only the weights

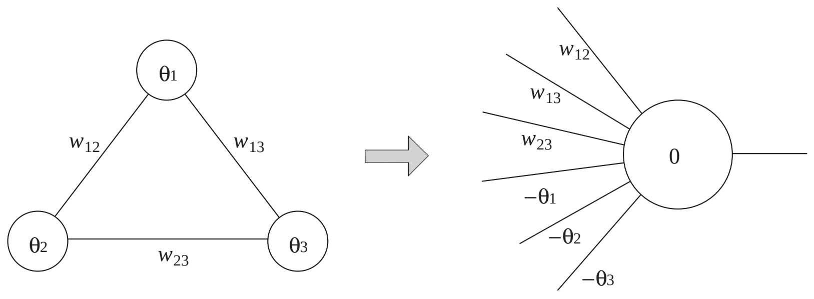


Fig. 13.14. Transformation of a Hopfield network into a perceptron

$w_{12}, w_{13}, \ldots, w_{1n}$ and the threshold θ_1 must be updated. This means that it is possible to use perceptron learning or the delta rule locally. During training all units are set to the desired stable states. If the sign of a unit's excitation is incorrect for the desired state, then the weights and threshold of this individual perceptron are corrected in the usual manner. It is not necessary to transform the Hopfield states into the $n(n+1)/2$-dimensional perceptron states every time we want to start the learning algorithm. This is only needed to prove the equivalence of Hopfield and perceptron learning.

13.4.2 Complexity of learning in Hopfield models

The interesting result which can immediately be inferred from the equivalence of Hopfield networks and perceptrons is that every learning algorithm for perceptrons can be transformed into a learning method for Hopfield networks. The delta rule or algorithms that proceed by finding inner points of solution polytopes can also be used to train Hopfield networks.

We have already shown in Chap. 10 that learning problems for multilayer networks are in general *NP*-complete. However, some special architectures can be trained in polynomial time. We saw in Chap. 4 that the learning problem for Hopfield networks can be solved in polynomial time, because there are learning algorithms for perceptrons whose complexity grows polynomially with the number of training vectors and their dimension (for example, Karmarkar's algorithm). Since the transformation described in the previous section converts m desired stable states into nm vectors to be linearly separated, and since this can be done in polynomial time, it follows that the learning problem for Hopfield networks can be solved in polynomial time. In Chap. 6 we also showed how to compute an upper bound for the number of linearly separable functions. This upper bound, valid for perceptrons, is also valid for Hopfield networks, since the stable states must be linearly separable (for the equivalent perceptron). This equivalence simplifies computation of the capacity of a Hopfield network when it is used as an associative memory.

13.5 Parallel combinatorics

The networks analyzed in the previous sections can be used either to compute Boolean functions or as associative memories. Those recurrent networks for which an energy function of a certain form exists can be used to solve some difficult problems in the fields of combinatorics and optimization theory. Hopfield networks have been proposed for these kinds of tasks.

13.5.1 *NP*-complete problems and massive parallelism

Many complex problems can be solved in a reasonable length of time using multiprocessor systems and parallel algorithms. This is easier for tasks that can be divided into independent subproblems, which are then assigned to different processors. The solution to the original problem is obtained by collecting the partial results after they have been computed. However, many well-known and important problems cannot be split in this manner. The parallel processes must cooperate and exchange information, so that the programmer must include some synchronization primitives in the system. If synchronization consumes too many resources and too much time, the parallel system may become only marginally faster than a sequential one.

Hopfield networks do not need any kind of synchronization; they guarantee that a local minimum of the energy function will be reached. If an optimization problem can be written in an analytical form isomorphic to the Hopfield energy function, it can be solved by a Hopfield network. We can assume that every unit in the network is simulated by a small processor. The states of the units can be computed asynchronously by transmitting the current unit states from processor to processor. There is no need for expensive synchronization and the task is solved by a massively parallel system. This strategy can be applied to all those combinatorial problems for whose solution large mainframes have traditionally been used.

We now show how to "load" an optimization problem on a Hopfield network discussing some progressively complicated examples. In the next subsections we will use the usual coding (with 0 and 1) for binary vectors and not the bipolar coding used in the previous examples.

13.5.2 The multiflop problem

Assume that we are looking for a binary vector of dimension n whose components are all zero with the exception of a single 1. The Hopfield network that solves this problem when $n = 4$ is depicted in Figure 13.15. Whenever a unit is set to 1, it inhibits the other units through the edges with weight -2. If the network is started with all units set to zero, then the excitation of every unit is zero, which is greater than the threshold and therefore the first unit to be asynchronously selected will flip its state to 1. No other unit can change its state after this first unit has been set to 1. A stable state has been reached.

One may think of this network as a generalization of the flip-flop network for two-dimensional vectors.

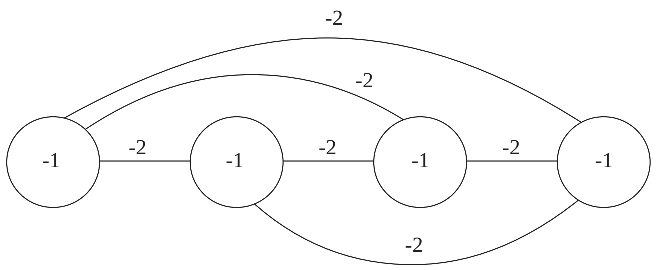


Fig. 13.15. A multiflop network

The weights for this network can be deduced from the following considerations. Let $x_1, x_2, \ldots, x_n$ denote the binary states of the individual units. Our task is to find a minimum of

$$E(x_1, \ldots, x_n) = (\sum_{i=1}^{n} x_i - 1)^2.$$

This expression can also be written as

$$E(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i^2 + \sum_{i \neq j}^{n} x_i x_j - 2 \sum_{i=1}^{n} x_i + 1.$$

For binary states it holds that $x_i = x_i^2$ and therefore

$$E(x_1, \ldots, x_n) = \sum_{i \neq j}^{n} x_i x_j - \sum_{i=1}^{n} x_i + 1$$

which can be rewritten as

$$E(x_1, \ldots, x_n) = -\frac{1}{2} \sum_{i \neq j}^{n} (-2) x_i x_j + \sum_{i=1}^{n} (-1) x_i + 1.$$

This expression is isomorphic to the energy function of the Hopfield network of Figure 13.15 (not considering the constant 1, which is irrelevant for the optimization problem). The network solves the multiflop problem in an automatic way by following its inherent dynamics.

13.5.3 The eight rooks problem

We make the optimization problem a notch more complicated: n rooks must be positioned in an $n \times n$ chess board so that no one figure can take another.

It is thus necessary to position each rook in a different row and column to the others. This problem can be thought of as a two-dimensional generalization of the multiflop problem. Each row is a chain of cells and only one of them can be set to 1. The same holds for each column.

The network of Figure 13.16 can solve this problem for a 4×4 board. Each field is represented by a unit. Only the connections of the first unit in the board are shown to avoid cluttering the diagram. The connections of each unit to all elements in the same row or column have the weight -2, all others have a weight zero. All units have the threshold -1. Any unit set to 1 inhibits any other units in the same row or column. If a row or column is all set to 0, when one of its elements is selected it will immediately switch its state to 1, since the total excitation (zero) is greater than the threshold -1.

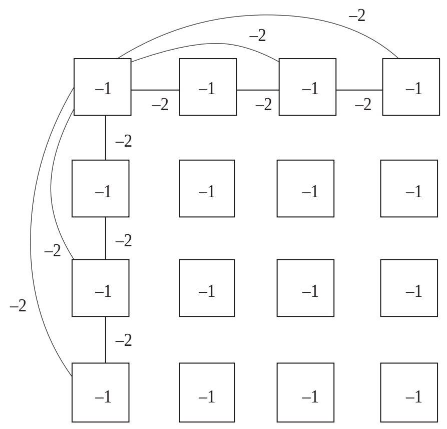


Fig. 13.16. Network for the solution of a four rooks problem

The weights for the network are derived from the following considerations: Let x_{ij} represent the state of the unit corresponding to the square ij in the $n \times n$ board. The number of ones in column j is given by $\sum_{i=1}^{n} x_{ij}$. If in each column only a single 1 is allowed, the following function must be minimized:

$$E_1(x_{11}, \ldots, x_{nn}) = \sum_{j=1}^{n} (\sum_{i=1}^{n} x_{ij} - 1)^2.$$

The minimum of the function corresponds to the situation in which just one rook has been positioned in every column. Similarly, for the rows of the board we define the function E_2 according to

$$E_2(x_{11}, \ldots, x_{nn}) = \sum_{i=1}^{n} (\sum_{j=1}^{n} x_{ij} - 1)^2.$$

We want to minimize the function $E = E_1 + E_2$. The general strategy is to reduce its analytical expression to a Hopfield form. The necessary algebraic steps can be avoided by noticing that the expression for E_1 is the sum of n independent functions (one per column). The term $(\sum_{i=1}^{n} x_{ij} - 1)^2$ corresponds to a multiflop problem. The weights for the edges in each column can be set to -2, as was done before in the multiflop problem. The same is done for each row: the weights between any unit and its row partners are set to -2. Only the thresholds must be selected with a little more care. The simple juxtaposition of a row-multiflop with a column-multiflop at each field will give us a threshold of $-1 + (-1) = -2$. This would mean that each row or column can contain up to two elements whose state is 1. This is avoided by setting the thresholds of the units to -1. The resulting network is the one shown in Figure 13.16. Each field will be forced to adopt the state zero whenever another unit is set to 1 in its own row or its own column.

13.5.4 The eight queens problem

The well-known eight queens problem can also be solved with a Hopfield network. It is just a generalization of the rooks problem, since now the diagonals of the board will also be considered. Each diagonal can be occupied at most once by a queen. As before with the rooks problem, we solve this task by overlapping multiflop problems at each square. Figure 13.17 shows how three multiflop chains have to be considered for each field. The diagram shows a 4×4 board and the overlapping of multiflop problems for the upper left square on the board. This overlapping provides us with the necessary weights, which are set to $w_{ij} = -2$, when unit i is different from unit j and belongs to the same row, column or diagonal as unit j. Otherwise we set w_{ij} to zero. The thresholds of all units are set to -1.

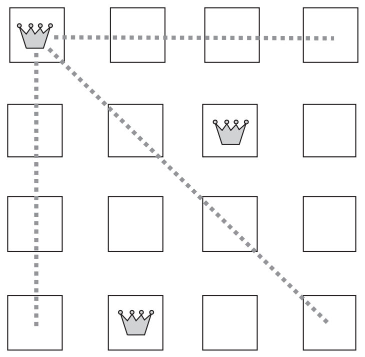


Fig. 13.17. The eight queens problem

A computer simulation shows, however, that this simple connection pattern does not always provide a correct solution for the n-queens problem. The proposed connection weights produce an energy function in which local minima with less than n queens are possible. The only alternative if such a stable state is found is to start the simulation again.

It is not possible to set the weights of the network of Figure 13.17 in such a way as to obtain only correct solutions. The difficulty is that diagonals can be occupied by a queen but they can also be unoccupied. Simple overlapping of multiflop problems does not work any more. The energy function has become much more complex than in the previous cases and we now have to achieve compromises between the weights which we choose for rows and columns and for diagonals.

13.5.5 The traveling salesman

The Traveling Salesman Problem (TSP) is one of the most celebrated benchmarks in combinatorics. It is simple to state and visualize and it is *NP*-complete. If we can solve the TSP efficiently, we can provide an efficient solution for other problems in the class *NP*. Hopfield and Tank [200] were the first to try to solve the TSP using a neural network.

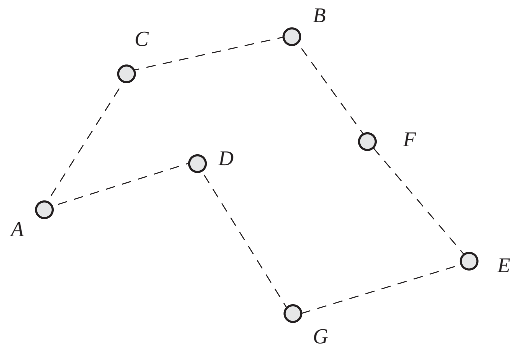


Fig. 13.18. A TSP and its solution

In the TSP we are looking for a path through n cities $S_1, S_2, \ldots, S_n$, such that every city is visited at least once and the length of a round trip is minimal. Let d_{ij} denote the distance between the cities S_i and S_j. A round trip can be represented using a matrix. Each of the n rows of the matrix is associated with a city. The columns are labeled 1 through n and they correspond to the n necessary visits. The following matrix shows a path going through the cities S_1, S_2, S_3 and S_4 in that same order:

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline S_1 & 1 & 0 & 0 & 0 \\ S_2 & 0 & 1 & 0 & 0 \\ S_3 & 0 & 0 & 1 & 0 \\ S_4 & 0 & 0 & 0 & 1 \end{array}$$

A single 1 is allowed in each row and each column, otherwise the salesman would visit a city twice or two cities simultaneously. This matrix must fulfill the same conditions as in the rooks problem and we can again use a network in which a unit is used to represent each entry in the matrix (the new "chess board").

Solving the TSP requires minimizing the length of the round trip, that is of

$$L = \frac{1}{2}\sum_{i,j,k}^{n} d_{ij} x_{ik} x_{j,k+1},$$

where x_{ik} represents the state of the unit corresponding to the entry ik in the matrix. When x_{ik} and $x_{j,k+1}$ are both 1, this means that the city S_i is visited in the k-th step and the city S_j in the step $k+1$. The distance between both cities must be added to the total length of the round trip. We use the convention that the column $n+1$ is equal to the first column of the matrix, so that we always obtain a round trip.

In the minimization problems we must include the constraints for a valid round trip. It is necessary to add the energy function of the rooks problem to the length L to obtain the new energy function E, which is given by

$$E = \frac{1}{2}\sum_{i,j,k}^{n} d_{ij} x_{ik} x_{j,k+1} + \frac{\gamma}{2}\Big(\sum_{j=1}^{n}\big(\sum_{i=1}^{n} x_{ij} - 1\big)^2 + \sum_{i=1}^{n}\big(\sum_{j=1}^{n} x_{ij} - 1\big)^2\Big),$$

where the constant γ regulates how much weight is given to the minimization of the length or to the generation of legal paths. The first summation to the right of the equal sign already has the form of a Hopfield energy function; the expression in parentheses has it too, since it is the energy function of a rooks problem. The weights for the network can be deduced immediately from this expression: the weights of edges between units in the same row or column must be set to $-\gamma$ and the thresholds of the units to $-\gamma/2$. The weights must be modified by including the length between states, so that the weight of the edge between unit ik and unit $j, k+1$ becomes

$$w_{ik,jk+1} = -d_{ij} + t_{ik,jk+1}$$

where $t_{ik,jk+1} = -\gamma$ whenever the units belong to the same row or column, otherwise $t_{ik,jk+1}$ is zero.

With this network we can try to find solutions of the TSP. A simulation shows that the generated routes are sometimes not legal, because one city is not visited or more than one city is visited in a single step. We can always force the network to generate legal tours: it is only necessary to set γ to a very large value so as to obliterate the contribution of the cities' distances. If γ is zero, we do not care whether the tour is a legal one and only the total length is minimized (by choosing an "empty" tour). Since the value of γ is not prescribed, it can be found by trial and error. In general the network will not be capable of finding the global minimum and because large TSPs (with more than 100 cities) have so many local minima, it is difficult to decide whether the local minimum that has been found is a good approximation to the optimal result. The whole approach is dependent on the existence of real, massively parallel systems, since the number of units required to solve a TSP increases quadratically with the number n of cities (and the number of weights increases proportionally to n^4).

13.5.6 The limits of Hopfield networks

The first articles of Hopfield and Tank on parallel solutions to combinatorial problems received a lot of attention [200, 201]. The theoretical question was whether this could be a method to solve *NP*-hard problems or at least to get an approximate solution in polynomial time. In the following years many other researchers tried to extend the range of combinatorial problems that could be solved using Hopfield's technique, trying to improve the quality of the results at the same time. It emerged that well-behaved average problems could be solved efficiently. However, these average results should be compared to the expected running time for the worst case. Bruck and Goodman [74] showed that a polynomially bounded network (on the size of the problem) is unable to find the global minimum of the energy function of *NP*-complete problems (encoded as Hopfield networks) with a 100% guarantee. Stated in another way: if we try to transform all local minima of the Hopfield network into an optimal solution of the combinatorial problem, the size of the network explodes exponentially. We proceed to prove the result of Bruck and Goodman, but we must first introduce an additional complexity class: the complement of the class *NP*.

The class *NP* of nondeterministic problems solvable in polynomial time is different from the class P of problems solvable in polynomial time not only in the way exposed already in Chap. 10. If a problem is a member of the class P, the same is true for the complementary problem. The complement of the decision problem "For the problem instance I, is X true for I?" is just "For the problem instance I, is X false for I?". A deterministic polynomial time algorithm terminates on each of the two questions. It is only necessary to substitute "true" for "false" to transform a polynomial time algorithm for a problem in P in an algorithm for its complement. But this is not necessarily so for problems in *NP*. A solution for the *Traveling Salesman Decision Problem* (TSDP), that is, the computation of the tour's length and the comparison with the decision's threshold, can be verified in polynomial time. However, the complementary problem has the wording "Is there no tour with a total length smaller than R?". If the answer is "yes", no polynomial time algorithm is known that could verify this assertion. It would be necessary to propose a data structure on which to perform some computations which could convince us of the truth of the assertion. Theoreticians assume that the complement of the TSDP probably does not belong to the class *NP*. The class which contains the complement of all *NP* problems is called co-*NP*. It is generally assumed that $NP \neq$ co-NP. This inequality is somewhat strong, since it implies that $P \neq NP$. Otherwise we would have co-$P =$ co-$NP = P = NP$, i.e., the equality $NP =$ co-NP would be valid. Yet theoreticians expect that eventually it will be proved that $NP \neq$ co-NP. Figure 13.19 illustrates the expected containments of the classes *NP*, *P* and co-*NP*.

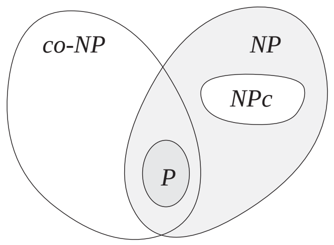


Fig. 13.19. The classes *NP* and co-*NP*

The following lemma determines under what conditions equality of the classes *NP* and co-*NP* would be possible. We can assume that this condition cannot be fulfilled.

Lemma 1. *If there is an* NP*-complete problem X whose complement X^c belongs to* NP*, then $NP =$ co-NP.*

The lemma is true because any problem Y in *NP* can be reduced in polynomial time into X. The complement of Y can therefore be transformed in polynomial time into X^c. Since a solution of X^c can be verified in polynomial time, the same is true for any solution of Y^c. This and some additional technical details would imply that $NP =$ co-NP.

Neural networks are just a subset of the algorithmic world. Since it is suspected that there is no polynomial time algorithm for the problems in the class *NP*, it should be possible to prove that Hopfield networks of bounded size are subjected to the same limitations. The following proposition settles this question [75].

Proposition 21. *Let L be an* NP*-complete decision problem and H a Hopfield network with a number of weights bounded by a polynomial on the size of the problem. If H can solve L (100% success rate) then $NP =$ co-NP.*

Proof. The problem L has a certain size defined by an appropriate coding. Since we must compute the energy function and from it derive the necessary weights for H, a polynomial bound on the total number of weights is necessary. A Hopfield network always finds a local minimum of its energy function. In our case, a 100% hit rate means that *all* local minima of the energy function should make possible a decision on the truth or falsity of the decision problem L. The Hopfield network can be considered a data structure that makes possible the verification of the found solution. It is only necessary to verify whether the solution found by the network is indeed a local minimum of the energy function. The polynomial size of the net makes this verification possible in polynomial time. The decision problem and the complement are, in this case, completely symmetric. The TSDP can be answered with "yes" if the tour found by the network is shorter than the decision threshold. But the complement of the TSDP can be decided also just by comparing the length of the optimal tour found with the decision threshold. Therefore the complement of L is a member of the class *NP* and it follows from Lemma 1 that $NP = \text{co-}NP$. Since it is generally assumed that this cannot be so, there should be a contradiction in the premises. The network H does not exist unless $NP = \text{co-}NP$. □

Even if we content ourselves with a polynomially bounded network that can provide approximate solutions (for example, TSP round-trips not larger by a given ε than the optimal tour), no such network can be built. It is because of this inherent limitation that some authors have sought to introduce stochastic factors into the networks, as we will discuss when we deal with *Boltzmann machines* in the next chapter.

13.6 Implementation of Hopfield networks

Hopfield networks as massively parallel systems are only interesting if they can be implemented in hardware and not just simulated in a sequential computer. Some proposals have been made for special chips capable of simulating Hopfield networks but the most promising approach are optical computers, capable of solving the connectivity problem of neural networks.

13.6.1 Electrical implementation

In 1984 Hopfield proposed an electrical realization of his model which uses a circuit very similar to the ones used by Steinbuch (Chap. 18) [199]. Figure 13.20 shows a diagram of the circuit. The outputs of the amplifiers $x_1, x_2, \ldots, x_n$ are interpreted as the states of the Hopfield units. Their complements $\neg x_1, \neg x_2, \ldots, \neg x_n$ are produced by inverters. All states and their complements are fed back to the input of the electrical circuit. An electrical contact is represented by a small circle. A resistance is present at each contact point. The connection r_{13}, for example, contains a resistor with resistance $r_{13} = 1/w_{13}$. The constants w_{ij} represent the weights of the Hopfield network between the units i and j. Inhibiting connections between one unit and another (that is, connections with negative weights) are simulated by connecting one inverted output of a unit to the other one. In Figure 13.20, for example, the connection points in the upper row all come from the inverted output of unit 1.

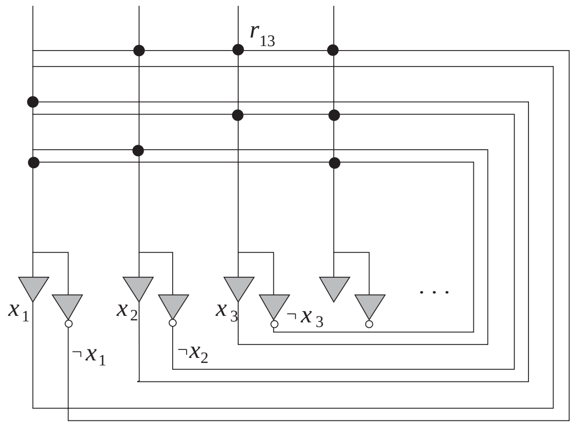


Fig. 13.20. Electrical implementation of a Hopfield network

In a network with n amplifiers the current I_i at the input to the i-th amplifier is given by

$$I_i = \sum_{j=1}^{n} \frac{x_j}{r_{ij}} = \sum_{j=1}^{n} x_j w_{ij},$$

where we have used the convention that r_{ij} is negative if the inverted value of x_j has been connected to the input of the amplifier x_i. The current I_i represents the excitation of unit i. The amplifier transforms the total excitation into 0 or 1 according to a certain electrical threshold, which can be set to an arbitrary positive value.

This simple circuit can be used to simulate Hopfield networks in a fraction of the time needed by a sequential computer. If the circuit is provided with variable resistors it is then possible to implement some learning algorithms directly in hardware.

13.6.2 Optical implementation

The most important computation that must be accelerated for a fast simulation of Hopfield networks is the vector matrix multiplication. Computation of the excitation of a node requires such an operation every time a unit's state is to be updated. Optical methods can be used to perform this numerical operation faster. Figure 13.21 shows an optical realization of the Hopfield model [132].

The logical structure is in principle the same as in the network of Figure 13.20, but the vector matrix multiplication is computed analogically using optical techniques. The n binary values which represent the network's state are projected through the vertical lens to the left of the arrangement. The lens projects each value x_i onto the corresponding row of an optical mask. Each row i in the mask is divided into fields which represent the n weights $w_{i1}, w_{i2}, \ldots, w_{in}$. Each field is partially darkened according to the value of the corresponding weight. The individual unit states are projected using light emitting diodes and the luminosity is proportional to the corresponding x_i value. The light going through the mask is collected by another lens in such a way that all the incoming light from a column is collected at a single position. The amount of light that goes through the mask is proportional to the product of x_i and w_{ij} at each position ij of the mask. The incoming light at the j-th detector represents the total excitation of unit j, which is equal to

$$s_j = \sum_{i=1}^{n} w_{ij} x_i.$$

The total excitation of the unit j can now be processed by an analog or digital circuit to produce the unit state which is used again in a new iteration of the network.

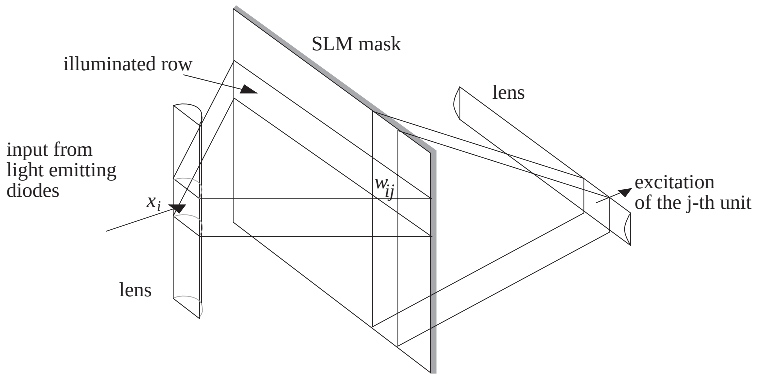


Fig. 13.21. Optical implementation of a Hopfield network

The difference compared with the electrical model is that the weights and signals must be normalized and scaled to fit the kind of optical processing being done. The most significant difference is the absence of direct connections. The light paths do not affect each other, so that it is possible to implement much larger networks than in the purely electrical realization. We will come back to the topic of optical implementations when we discuss neural hardware in Chap. 18.

13.7 Historical and bibliographical remarks

With the introduction in 1982 of the model named after him, John Hopfield established the connection between neural networks and physical systems of the type considered in statistical mechanics. This in turn gave computer scientists a whole new arsenal of mathematical tools for the analysis of neural networks. Other researchers had already considered more general associative memory models in the 1970s, but by restricting the architecture of the network to a symmetric connection matrix with a zero diagonal, it was possible to design recurrent networks with stable states. With the introduction of the concept of the energy function, the convergence properties of the networks could be more easily analyzed.

The Hopfield network also has the advantage, in comparison with other models, of a simple technical implementation using electronic or optical devices [132]. The computing strategy used when updating the network states corresponds to the relaxation methods traditionally used in physics [92].

The properties of Hopfield networks have been investigated since 1982 using the theoretical tools of statistical mechanics [322]. Gardner [155] published a classical treatise on the capacity of the perceptron and its relation to the Hopfield model. The total field sensed by particles with a spin can be computed using the methods of mean field theory. This simplifies a computation which is hopeless without the help of some statistical assumptions [189]. Using these methods Amit et al. [24] showed that the number of stable states in a Hopfield network of n units is bounded by $0.14n$. A *recall* error is tolerated only 5% of the time. This upper bound is one of the most cited numbers in the theory of Hopfield networks.

In 1988 Kosko proposed the BAM model, which is a kind of "missing link" between conventional associative memories and Hopfield networks. Many other variations have been proposed since, some of them with asynchronous, others with synchronous dynamics [231]. Hopfield networks have also been studied from the point of view of dynamical systems. In this respect spin glass models play a relevant role. These are materials composed of particles with a spin and mutual interactions [412].

Combinatorial problems have a long tradition, but a really systematic theory capable of unifying the myriad of heuristic methods developed in the past was first developed in the 1960s and 1970s [361]. The important point was the increasingly important role played by computers and the emergence of a new attitude which tried to reach whole classes of problems and not just individual cases. An important research theme which remains is how to split a combinatorial problem into subtasks that can be assigned to different processors [160].

The efforts of Hopfield and Tank with the TSP led to many other similar experiments in related fields. Wilson and Pawley [456] repeated their experiments but they could not confirm the optimistic results of the former authors. The main difficulty is that complex combinatorial problems produce an exponential number of local minima of the energy function. In sequential computers, Hopfield models cannot compete with conventional methods [224]. Many heuristics have been proposed for the TSP, starting with the classical work by Kernighan and Lin [274]. The only way to make Hopfield models competitive is through the use of special hardware. Sheu et al. have obtained interesting results and significant speedup in comparison with sequential computers by using a technique they call *hardware annealing*.

One of the first to deal with the intrinsic limits of the Hopfield model for the solution of the TSP was Abu-Mostafa [3], who nevertheless considered only the case of networks of constant size. Bruck and Goodman [75] considered networks of variable but polynomially bounded size and obtained the same negative results. Although this almost meant the "death of the traveling salesman" [322], the Hopfield model and its stochastic variants have been applied in many other fields, such as psychology, simulation of ensembles of biological neural networks, and chaotic behavior of neural circuits.

The optical implementation of Hopfield networks is a promising field of research. Other than masks, holograms can also be used to store the network weights [352]. The main technical problem is still the size reduction of the optical components, which could make them a viable alternative to conventional electronic systems.

Exercises

1. Train a Hopfield network in the computer using the perceptron learning algorithm.
2. Solve a TSP of 10 cities with a Hopfield network. How many weights do you need for the network?
3. Compute the energy of the different possible states of the network shown in Figure 13.6. Do the same for Figure 13.7 assigning some values to the weights and thresholds.

14

Stochastic Networks

14.1 Variations of the Hopfield model

In the previous chapter we showed that Hopfield networks can be used to provide solutions to combinatorial problems that can be expressed as the minimization of an energy function, although without guaranteeing global optimality. Once the weights of the edges have been defined, the network shows spontaneous computational properties. Harnessing this spontaneous dynamics for useful computations requires some way of avoiding falling into local minima of the energy function in such a way that the global minimum is reached. In the case of the eight queens problem, the number of local minima is much higher than the number of global minima and very often the Hopfield network does not stabilize at a correct solution. The issue to be investigated is therefore whether a certain variation of the Hopfield model could achieve better results in combinatorial optimization problems.

One possible strategy to improve the global convergence properties of a network consists in increasing the number of paths in search space, in such a way that not only binary but also real-valued states become possible. Continuous updates of the network state become possible and the search for minima of the energy function is no longer limited to the corners of a hypercube in state space. All interior points are now legal network states. A continuous activation function also makes it possible to describe more precisely the electrical circuits used to implement Hopfield networks.

A second strategy to avoid local minima of the energy function consists in introducing *noise* into the network dynamics. As usual, the network will reach states of lower energy but will also occasionally jump to states higher up in the energy ladder. In a statistical sense we expect that this extended dynamics can help the network to skip out of local minima of the energy function. The best-known models of such stochastic dynamics are *Boltzmann machines*.

First we will discuss real-valued states and continuous dynamics. The main difficulty in handling this extended model is providing a precise definition of the energy function in the continuous case.

14.1.1 The continuous model

The Hopfield model with binary or bipolar states can be generalized by accepting, just as in backpropagation networks, all real values in the interval $[0, 1]$ as possible unit states. Consequently, the *discrete* Hopfield model becomes the *continuous* model. The activation function selected is the sigmoid and the dynamics of the network remains asynchronous.

The difference between the new and the discrete model is given by the activation assumed by unit i once it has been asynchronously selected for updating. The new state is given by

$$x_i = s(u_i) = \frac{1}{1 + \mathrm{e}^{-u_i}}$$

where u_i denotes the net excitation of the unit. Additionally we assume that a unit's excitation changes slowly in time. Changes in the state of other units will not be registered instantaneously, but with a certain delay. The net excitation of unit i changes in time according to

$$\frac{du_i}{dt} = \gamma \left(-u_i + \sum_{j=1}^{n} w_{ij} x_j \right) = \gamma \left(-u_i + \sum_{j=1}^{n} w_{ij} s(u_j) \right), \tag{14.1}$$

where γ denotes a positive learning constant and w_{ij} the weight between unit i and unit j. For a simulation we compute a discrete approximation of du_i, which is added to the current value of u_i. The result leads to the new state $x_i = s(u_i)$.

We have to show that the defined dynamics of the continuous model leads to equilibrium. With this objective in mind we define an energy function with only slight differences from the one for the discrete model. Hopfield [199] proposes the following energy function:

$$E = -\frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} w_{ij} x_i x_j + \sum_{i=1}^{n} \int_{0}^{x_i} s^{-1}(x) dx.$$

We just have to show that the energy of the network becomes lower after each state update. The change in time of the energy is given by

$$\frac{dE}{dt} = -\sum_{i=1}^{n} \sum_{j=1}^{n} w_{ij} \frac{dx_i}{dt} x_j + \sum_{i=1}^{n} s^{-1}(x_i) \frac{dx_i}{dt}.$$

This expression can be simplified by remembering that the network is symmetric (i.e., $w_{ij} = w_{ji}$). Since $u_i = s^{-1}(x_i)$ it holds that

$$\frac{dE}{dt} = -\sum_{i=1}^{n} \frac{dx_i}{dt} \left(\sum_{j=1}^{n} w_{ij} x_j - u_i \right)$$

From equation (14.1) we now obtain

$$\frac{dE}{dt} = -\frac{1}{\gamma}\sum_{i=1}^{n} \frac{dx_i}{dt}\frac{du_i}{dt}.$$

Since $x_i = s(u_i)$, the above expression can be written as

$$\frac{dE}{dt} = -\frac{1}{\gamma}\sum_{i=1}^{n} s'(u_i)\left(\frac{du_i}{dt}\right)^2.$$

We know that $s'(u_i) > 0$ because the sigmoid is a strict monotone function, and since the constant γ is also positive we finally conclude that

$$\frac{dE}{dt} \leq 0.$$

The defined dynamics of the network implies that the energy is reduced or remains unchanged after each update. A stable state is reached when dE/dt vanishes. This happens when du_i/dt reaches the saturation region of the sigmoid where $du_i/dt \approx 0$. The state of all units can no longer change appreciably.

The time needed for convergence can increase exponentially, since du_i/dt gets smaller and smaller. We must therefore require convergence to a neighborhood of a local minimum and this can happen in finite time. Some authors have proposed variations of (14.1) that accelerate the convergence process.

Experimental results show that the continuous Hopfield model can find better solutions for combinatorial problems than the discrete model. However, in the case of really hard problems (for example the TSP), the continuous model has no definite advantage over the discrete model. Still, even in this case the continuous model remains interesting, because it can be more easily implemented with analog hardware.

14.2 Stochastic systems

The optimization community has been using a method known as *simulated annealing* for many years. [304]. Annealing is a metallurgical process in which a material is heated and then slowly brought to a lower temperature. The crystalline structure of the material can reach a global minimum in this way. The high temperature excites all atoms or molecules, but later, during the cooling phase, they have enough time to assume their optimal position in the crystalline lattice and the result is fewer fractures and fewer irregularities in the crystal.

Annealing can avoid local minima of the lattice energy because the dynamics of the particles contains a temperature-dependent component. The particles not only lose energy during the cooling phase, but sometimes borrow some energy from the background and assume a higher-energy state. Shallow

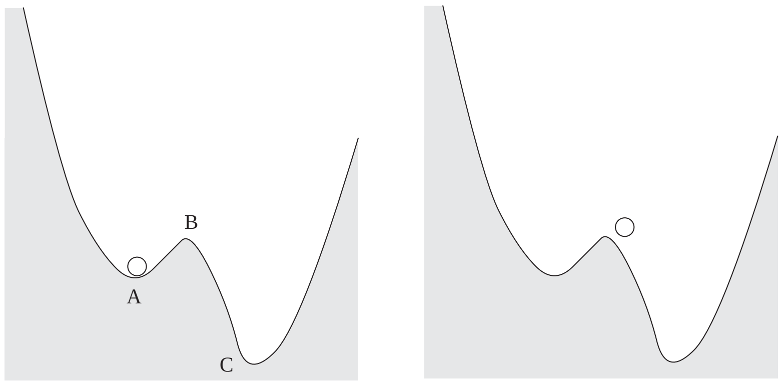


Fig. 14.1. The effect of thermal noise

minima of the energy function can be avoided in this way. This situation is shown in Figure 14.1. If the dynamics of the system is such that the system converges to local energy minima, the system state can be trapped at position A, although C is a more favorable state in terms of the global energy. Some thermal noise, that is, local energy fluctuations, can give the system the necessary impulse to skip over energy barrier B and reach C.

14.2.1 Simulated annealing

To minimize a function E we simulate this phenomenon in the following way: the value of the free variable x is changed always if the update Δx can reduce the value of the function E. However, if the increment actually increases the value of E by ΔE, the new value for x, that is $x + \Delta x$ is accepted with probability

$$p_{\Delta E} = \frac{1}{1 + \mathrm{e}^{\Delta E/T}},$$

where T is a temperature constant. For high values of T, $p_{\Delta E} \approx 1/2$ and the update is accepted half the time. If $T = 0$ only those updates which reduce the value of E are accepted with probability 1. Varying the value of T, from large values all the way down to zero, corresponds to the heating and cooling phases of the annealing process. This explains why this computational technique has been called simulated annealing.

We expect that simulated annealing will allow us to simulate the dynamics of Hopfield networks in such a way that deeper regions of the energy function are favored. It can even be shown that with this simulation strategy the global minimum of the energy function is asymptotically reached [1]. Many other kinds of optimization algorithms can be combined with an annealing phase in order to improve their results. The sigmoid is used for the computation of the probability because it corresponds to those functions used in thermodynamics for the analysis of thermal equilibrium. This simplifies the statistical analysis of the method, because it is possible to use some results known from statistical mechanics.

Hopfield networks can be considered as optimization systems and thermal noise can be introduced in different ways in the network dynamics. We can distinguish between *Gauss* or *Boltzmann machines*, depending on the stage at which noise is introduced into the computation.

14.2.2 Stochastic neural networks

The dynamics of a Hopfield network can be generalized according to the strategy described in our next definition.

Definition 18. *A Boltzmann machine is a Hopfield network composed of n units with states $x_1, x_2, \ldots, x_n$. The state of unit i is updated asynchronously according to the rule*

$$x_i = \begin{cases} 1 & \textit{with probability } \; p_i, \\ 0 & \textit{with probability } 1 - p_i, \end{cases}$$

where

$$p_i = \frac{1}{1 + \exp(-(\sum_{j=1}^{n} w_{ij} x_j - \theta_i)/T)}$$

and T is a positive temperature constant. The constants w_{ij} denote the network weights and the constants θ_i the bias of the units.

The Boltzmann machine can be defined using bipolar or binary states, but in this chapter we will stick to binary states. The energy function for a Boltzmann machine has the same form as for Hopfield networks:

$$E = -\frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} w_{ij} x_i x_j + \sum_{i=1}^{n} \theta_i x_i$$

The difference between a Boltzmann machine and a Hopfield network is the stochastic activation of the units. If T is very small then $p_i \approx 1$ when $\sum_{i=1}^{n} w_{ij} - \theta_i$ is positive. If the net excitation is negative, then $p_i \approx 0$. In this case the dynamics of the Boltzmann machine approximates the dynamics of the discrete Hopfield network and the Boltzmann machine settles on a local minimum of the energy function. If $T > 0$, however, the probability of a transition, or a sequence of transitions, from a network state $x_1, x_2, \ldots, x_n$ to another state is never zero. The state space of the Boltzmann machine contains the 2^n vertices of the n-dimensional hypercube. When a network state is selected, we are choosing a vertex of the hypercube. The probability of reaching any other vertex that is different only at the i-th single component is proportional to p_i when $x_i = 0$ or to $1 - p_i$ when $x_i = 1$. In both cases the probability of a transition does not vanish. A transition from any given vertex to another is always possible. The transition probability along a predefined path is given by the product of all the transition probabilities from vertex to vertex along this path. The product is positive whenever $T > 0$. Therefore, Boltzmann machines with $T > 0$ cannot settle on a single state; they reduce the energy of the system but they also let it become larger. During the cooling phase we expect those transitions from higher to lower energy levels to be more probable, as it happens in simulated annealing experiments. When T is large, the network state visits practically the complete state space. During the cooling phase, the network begins to stay longer in the basins of attraction of the local minima. If the temperature is reduced according to the correct schedule, we can expect the system to reach a global minimum with the integrated transition probability 1.

14.2.3 Markov chains

Let us illustrate the dynamics defined in the last section with a small example. Take one of the smallest Hopfield networks, the flip-flop. The weight between the two units is equal to -1 and the threshold is -0.5.

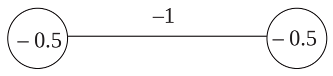


Fig. 14.2. A flip-flop network

The network works using binary states. There are four of them with the following energies:

$$\begin{array}{llll} E_{00} = & 0.0 & E_{10} = & -0.5 \\ E_{01} = & -0.5 & E_{11} = & 0.0 \end{array}$$

Figure 14.3 shows the distribution of the four states in energy levels. We have included the transition probabilities from one state to the other, under the assumption $T = 1$. For example, for a transition from state 00 to state 01, the second unit must be selected and its state must change from 0 to 1. The probability of this happening is

$$p_{00\to 01} = \frac{1}{2}\left(\frac{1}{1+\exp(-0.5)}\right) = 0.31$$

The factor $1/2$ is used in the computation because any one of two units can be selected in the update step and the probability that a unit is selected for an update is $1/2$. The argument 0.5 corresponds to the net excitation of the second unit in the network state 00. All other transition probabilities were computed in a similar way.

In order to analyze the dynamics of a Boltzmann machine we need to refer to its matrix of transition probabilities.

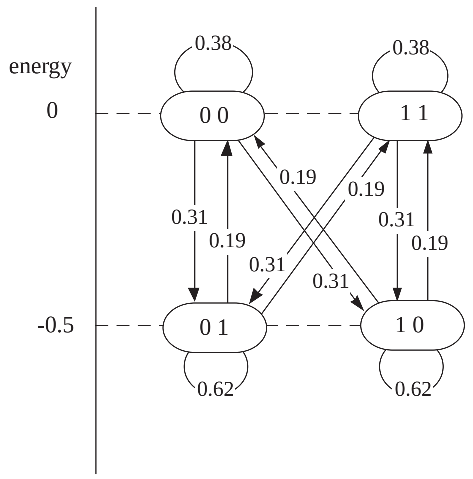


Fig. 14.3. Transition probabilities of a flip-flop network ($T = 1$)

Definition 19. *The state transition matrix $\mathbf{P}_T = \{p_{ij}\}$ of a Boltzmann machine with n units is the $2^n \times 2^n$ matrix, whose elements p_{ij} represent the probability of a transition from state i to the state j in a single step at the temperature T.*

$\mathbf{P}_T$ is a $2^n \times 2^n$ matrix because there are 2^n network states. The transition matrix is, according to the definition, a sparse matrix. The matrix for the flip-flop network, for example, is of dimension 4×4. The diagonal elements p_{ii} of the matrix $\mathbf{P}_T$ represent the probabilities that state i does not change. If the states 00, 01, 10, and 11 are taken in this order, the matrix $\mathbf{P} \equiv \mathbf{P}_1$ for the flip-flop network is given by:

$$\mathbf{P} = \begin{pmatrix} \frac{1}{1+e^{1/2}} & \frac{1}{2}\frac{1}{1+e^{-1/2}} & \frac{1}{2}\frac{1}{1+e^{-1/2}} & 0 \\ \frac{1}{2}\frac{1}{1+e^{1/2}} & \frac{1}{1+e^{-1/2}} & 0 & \frac{1}{2}\frac{1}{1+e^{1/2}} \\ \frac{1}{2}\frac{1}{1+e^{1/2}} & 0 & \frac{1}{1+e^{-1/2}} & \frac{1}{2}\frac{1}{1+e^{1/2}} \\ 0 & \frac{1}{2}\frac{1}{1+e^{-1/2}} & \frac{1}{2}\frac{1}{1+e^{-1/2}} & \frac{1}{1+e^{1/2}} \end{pmatrix}.$$

The numerical value of the matrix in our example is:

$$\mathbf{P} = \begin{pmatrix} 0.38 & 0.31 & 0.31 & 0 \\ 0.19 & 0.62 & 0 & 0.19 \\ 0.19 & 0 & 0.62 & 0.19 \\ 0 & 0.31 & 0.31 & 0.38 \end{pmatrix}.$$

It follows from the definition of the transition matrix, that $0 \leq p_{ij} \leq 1$. As long as $T > 0$, it holds that $0 \leq p_{ij} < 1$, since no network state is stable. It should be clear that

$$\sum_{j=1}^{n} p_{ij} = 1,$$

because at each step the state i is changed with probability $\sum_{j \neq i}^{n} p_{ij}$ or remains the same with probability p_{ii}. A matrix with elements $p_{ij} \geq 0$ and whose rows add up to 1 is called a *stochastic matrix.* Boltzmann machines are examples of a *first-order Markov process*, because the transition probabilities only depend on the current state, not on the history of the system. In this case the matrix is called a *Markov matrix.* Using the transition matrix it is possible to compute the probability distribution of the network states at any time t. Assume, for example, that each of the four states of the flip-flop network is equiprobable at the beginning. The original probability distribution $\mathbf{v}_0$ is therefore

$$\mathbf{v}_0 = (0.25,\, 0.25,\, 0.25,\, 0.25)$$

The probability distribution $\mathbf{v}_1$ after an update step of the network is given by

$$\begin{aligned}
\mathbf{v}_1 &= \mathbf{v}_0 P \\
&= (0.25,\, 0.25,\, 0.25,\, 0.25) \begin{pmatrix} 0.38 & 0.31 & 0.31 & 0 \\ 0.19 & 0.62 & 0 & 0.19 \\ 0.19 & 0 & 0.62 & 0.19 \\ 0 & 0.31 & 0.31 & 0.38 \end{pmatrix} \\
&= (0.19,\, 0.31,\, 0.31,\, 0.19)
\end{aligned}$$

As can be seen, for $T = 1$ the flip-flop network assumes the states 01 and 10 62% of the time, and 38% of the time the states 00 and 11. The development of the probability distribution of the network states is computed in general according to

$$\mathbf{v}_t = \mathbf{v}_{t-1}\mathbf{P}.$$

A dynamical system obeying such an equation in which a Markov matrix $\mathbf{P}$ is involved, is called a *Markov chain.* The main issue is finding the final probability distribution given by a vector $\mathbf{v}$ for which

$$\mathbf{v} = \mathbf{v}\mathbf{P}\,.$$

The vector $\mathbf{v}$ is therefore an eigenvector of $\mathbf{P}$ with eigenvalue 1. A fundamental result of Markov chains theory states that such a stable distribution $\mathbf{v}$ always exists, if all states can be reached from any other state in one or more steps, and with non-zero probability [363]. Since all Boltzmann machines with $T > 0$ fulfill these conditions, such a stable probability distribution, representing *thermal equilibrium* of the network dynamics, exists. It is important to underline that the network arrives to the stable distribution independently of the initial one. If the initial distribution is denoted by $\mathbf{v}_0$, then

$$\mathbf{v}_t = \mathbf{v}_{t-1}\mathbf{P} = \mathbf{v}_{t-2}\mathbf{P}^2 = \cdots = \mathbf{v}_0\mathbf{P}^t$$

When the stable state $\mathbf{v}$ has been reached, the matrix $\mathbf{P}^T$ has stabilized. In the case of the flip-flop network, the matrix $\mathbf{P}^t$ converges to

$$\lim_{t\to\infty} \mathbf{P}^t = \begin{pmatrix} 0.19 & 0.31 & 0.31 & 0.19 \\ 0.19 & 0.31 & 0.31 & 0.19 \\ 0.19 & 0.31 & 0.31 & 0.19 \\ 0.19 & 0.31 & 0.31 & 0.19 \end{pmatrix}.$$

The stable matrix consists of four identical rows. Starting from an arbitrary distribution (a_1, a_2, a_3, a_4) we can compute the stable distribution:

$$\begin{aligned} \mathbf{v} = (a_1, a_2, a_3, a_4) & \begin{pmatrix} 0.19 & 0.31 & 0.31 & 0.19 \\ 0.19 & 0.31 & 0.31 & 0.19 \\ 0.19 & 0.31 & 0.31 & 0.19 \\ 0.19 & 0.31 & 0.31 & 0.19 \end{pmatrix} \\ = (0.19,\ 0.31,\ 0.31,\ 0.19). & \end{aligned}$$

In the computation we made use of the fact that $a_1 + a_2 + a_3 + a_4 = 1$, since this must hold for all probability distributions. The same distribution was found in the vector-matrix multiplication after a single step (because of the simplicity of the problem). As this example has shown, any Boltzmann machine can be analyzed by constructing the transition matrix in order to find its eigenvectors using any of the common iterative methods.

14.2.4 The Boltzmann distribution

It is theoretically possible to use probability functions other than the sigmoid in order to define the network dynamics. Its advantage, nevertheless, is that we can find this kind of transition function in many interesting physical processes.

Consider a system in thermal equilibrium with m different states and associated energies $E_1, E_2, \ldots, E_m$. The probability p_{ij} of a transition from a state of energy E_i to another state of energy E_j is given by

$$p_{ij} = \frac{1}{1 + \mathrm{e}^{(E_j - E_i)/T}} \tag{14.2}$$

Physical systems obeying such a probability function for energy transitions, and at thermal equilibrium, reach a stable probabilistic state known as the *Boltzmann distribution.*

According to the Boltzmann distribution the probability p_i that a system assumes the energy level E_i during thermal equilibrium is

$$p_i = \frac{\mathrm{e}^{-E_i/T}}{Z} \tag{14.3}$$

In this formula $Z = \sum_{i=1}^{m} \mathrm{e}^{-E_i/T}$ is a normalizing factor known as the state sum.

It is important to deal with the Boltzmann distribution analytically because it provides us with the solution of the eigenvector problem for a Markov matrix without having to perform the iterative computations described before. The precise derivation of the Boltzmann distribution is done in ergodic theory, which deals with the dynamics of infinite Markov chains, among other problems.

We can nevertheless verify the validity of the Boltzmann distribution with some straightforward calculations. For a system in thermal equilibrium, the transition matrix for m states with associated energies $E_1, E_2, \ldots, E_m$ is the following:

$$P = \begin{pmatrix} 1 - \sum\limits_{i=2}^{m} \frac{1}{1+\mathrm{e}^{(E_i-E_1)/T}} & \cdots & \frac{1}{1+\mathrm{e}^{(E_m-E_1)/T}} \\ \frac{1}{1+\mathrm{e}^{(E_1-E_2)/T}} & \ddots & \vdots \\ \vdots & & \vdots \\ \frac{1}{1+\mathrm{e}^{(E_1-E_m)/T}} & \cdots & 1 - \sum\limits_{i=1}^{m-1} \frac{1}{1+\mathrm{e}^{(E_i-E_m)/T}} \end{pmatrix}$$

The product of the Boltzmann distribution

$$\frac{1}{Z}\left(\mathrm{e}^{-E_1/T}, \mathrm{e}^{-E_2/T}, \ldots, \mathrm{e}^{-E_m/T}\right).$$

with $\mathbf{P}$ is a vector of dimension m. Its first component is

$$\begin{aligned} v_1 &= \frac{1}{Z}\left(\mathrm{e}^{-E_1/T}\left(1 - \sum_{i=2}^{m} \frac{1}{1+\mathrm{e}^{(E_i-E_1)/T}}\right) + \sum_{i=2}^{m} \frac{\mathrm{e}^{-E_i/T}}{1+\mathrm{e}^{(E_1-E_i)/T}}\right) \\ &= \frac{1}{Z}\left(\mathrm{e}^{-E_1/T} - \sum_{i=2}^{m} \frac{\mathrm{e}^{-E_1/T}}{1+\mathrm{e}^{(E_i-E_1)/T}} + \sum_{i=2}^{m} \frac{\mathrm{e}^{-E_i/T}}{1+\mathrm{e}^{(E_1-E_i)/T}}\right) \\ &= \frac{1}{Z}\left(\mathrm{e}^{-E_1/T} - \sum_{i=2}^{m} \frac{1}{\mathrm{e}^{E_1/T}+\mathrm{e}^{E_i/T}} + \sum_{i=2}^{m} \frac{1}{\mathrm{e}^{E_i/T}+\mathrm{e}^{E_1/T}}\right) \\ &= \frac{1}{Z}\mathrm{e}^{-E_1/T}. \end{aligned}$$

We again get the first component of the Boltzmann distribution. The same can be shown for the other components, proving that the distribution is stable with respect to $\mathbf{P}$.

It easy to show that a Boltzmann machine is governed by equation (14.2). If a transition from state α to state β occurs, a unit k must have changed its state from x_k to x'_k. The energy of the network in state α is given by

$$E_\alpha = -\frac{1}{2}\sum_{i\neq k}^{n}\sum_{j\neq k}^{n} w_{ij}x_i x_j + \sum_{i\neq k}^{n}\theta_i x_i - \sum_{i=1}^{n} w_{ki}x_i x_k + \theta_k x_k,$$

in state β by

$$E_\beta = -\frac{1}{2}\sum_{i\neq k}^{n}\sum_{j\neq k}^{n} w_{ij}x_i x_j + \sum_{i\neq k}^{n}\theta_i x_i - \sum_{i=1}^{n} w_{ki}x_i x'_k + \theta_k x'_k.$$

The contribution of unit k was expressed in both cases separately. The energy difference is therefore

$$E_\beta - E_\alpha = -\sum_{i=1}^{n} w_{ki}x_i(x'_k - x_k) + \theta_k(x'_k - x_k). \tag{14.4}$$

Since the states are different $x_k \neq x'_k$. Assume without loss of generality that $x_k = 0$ and the unit state changed to $x'_k = 1$. This happened following the defined rules, that is, with probability

$$p_{\alpha\to\beta} = \frac{1}{1+\exp\left(-\left(\sum_{i=1}^{n} w_{ki}x_i - \theta_k\right)\right)}. \tag{14.5}$$

Since $x'_k = 1$ it holds that $x'_k - x_k = 1$ and (14.4) transforms to

$$E_\beta - E_\alpha = -\sum_{i=1}^{n} w_{ki}x_i + \theta_k. \tag{14.6}$$

The transition is ruled by equation (14.5), which is obviously equivalent to (14.2). The only limitation is that in Boltzmann machines state transitions involving two units are not allowed, but this does not affect the substance of our argument. The Boltzmann distribution is therefore the stable probability distribution of the network. In our flip-flop example, the stable distribution is given by

$$\begin{aligned}\mathbf{v} &= \frac{1}{\mathrm{e}^{-E_{00}} + \mathrm{e}^{-E_{01}} + \mathrm{e}^{-E_{10}} + \mathrm{e}^{-E_{11}}}\left(\mathrm{e}^{-E_{00}}, \mathrm{e}^{-E_{01}}, \mathrm{e}^{-E_{10}}, \mathrm{e}^{-E_{11}}\right)\\ &= \frac{1}{2}\left(\frac{1}{1+\mathrm{e}^{0.5}}\right)\left(1, \mathrm{e}^{0.5}, \mathrm{e}^{0.5}, 1\right)\\ &= (0.19, 0.31, 0.31, 0.19)\,.\end{aligned}$$

The result is the same as that computed previously using the direct method.

14.2.5 Physical meaning of the Boltzmann distribution

It is interesting to give the Boltzmann distribution a physical interpretation. This can be done using an example discussed by Feynman. The atmosphere is a system in which the air molecules occupy many different energy levels in the gravitational field. We can investigate the distribution of the air molecules in an air column in thermal equilibrium. Figure 14.4 shows a slice of thickness dh of a vertical air cylinder. The slice is cut at height h. The weight G of a differential air cylinder with unitary base area is

$$G = -mgn\,dh$$

where m represents the mass of an air particle (all of them are equal to simplify the derivation), g the gravitational field strength, and n the number of air particles in a unitary volume. In that case, $n\,dh$ is the number of particles in a differential cylinder.

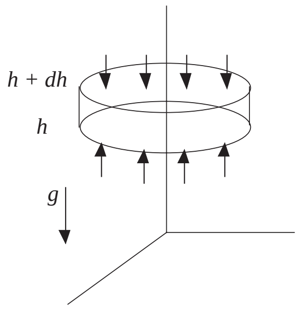


Fig. 14.4. Horizontal slice of an air column in thermal equilibrium

The weight G is equal to the difference between the pressure P_{h+dh} on the upper side of the cylinder and the pressure P_h on the lower side, that is,

$$dP = P_{h+dh} - P_h = -mgn\,dh$$

According to the theory of ideal gases,

$$P = nkT,$$

where k represents the Boltzmann constant and T the temperature. Consequently, $dP = kT dn$. Combining the last two equations we get

$$\begin{aligned} kT\,dn &= -mgn\,dh \\ \frac{dn}{n} &= -\frac{mg}{kT}dh \end{aligned}$$

From this we conclude that

$$n = n_0 \mathrm{e}^{-mgh/kT}.$$

The equation tells us that the number of air particles decreases exponentially with altitude (and correspondingly with the energy level). The equation is similar to (14.3). The above analysis shows therefore that the Boltzmann distribution determines the mean density of the particles which reach the energy level $-mgh/kT$. We can deduce similar conclusions for other physical systems governed by the Boltzmann distribution.

The above result gives an intuitive idea of one of the essential properties of the Boltzmann distribution: a system in thermal equilibrium stays in states of low energy with greater probability. Fluctuations can occur, but the number of transitions to higher energy levels decreases exponentially. Those transitions can be increased by raising the temperature T. The system is "heated" and the number of state transitions grows accordingly.

14.3 Learning algorithms and applications

As we have shown, when the temperature T is positive there are no absolutely stable states. Nevertheless we can ask if a Boltzmann machine can be trained to reproduce a given probability distribution of network states. If this can be done the network will learn to behave in a statistically reproducible way, once some of the inputs have been fixed.

14.3.1 Boltzmann learning

Ackley, Hinton, and Sejnowski developed the algorithm called *Boltzmann learning* [7]. We follow Hertz et al. in the derivation [189]. An alternative proof can be found in [108].

The states of the input units of the network will be indexed with α and the states of the hidden units with β (hidden units are those which receive no external input). If there are m input and k hidden units, the state of the network is specified by the state of $n = m+k$ units. Without loss of generality we consider only units with threshold zero.

The probability P_α that the input units assume state α (without considering the state β of the hidden units) is given by

$$P_\alpha = \sum_\beta P_{\alpha\beta} \tag{14.7}$$

where $P_{\alpha\beta}$ denotes the probability of the combined network state $\alpha\beta$ and the sum is done considering all possible states of the hidden units. Equation (14.7) can be written as

$$P_\alpha = \frac{1}{Z} \sum_\beta \exp(-\gamma E_{\alpha\beta}) \tag{14.8}$$

where

$$Z = \sum_{\alpha,\beta} \exp(-\gamma E_{\alpha\beta}) \tag{14.9}$$

and $\gamma = 1/T$. Equations (14.8) and (14.9) are derived from the Boltzmann distribution. The energy of the network in state $\alpha\beta$ is

$$E_{\alpha\beta} = -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij} x_i^{\alpha\beta} x_j^{\alpha\beta}. \tag{14.10}$$

Equation (14.7) provides us with the probability distribution of the states α in a *free running* network, that is, a network without external input. We want the network to learn to take the states α with the probability distribution R_α. During the training phase, an input vector is clamped to the input units. The distance D between the desired and the actual behavior of the network is measured using the expression

$$D = \sum_{\alpha} R_\alpha \log \frac{R_\alpha}{P_\alpha}. \tag{14.11}$$

The distance D is zero only if the distribution R_α is equal to the distribution P_α. The learning algorithm must minimize D. Gradient descent on this error function leads to the weight update

$$\Delta w_{ij} = -\eta \frac{\partial D}{\partial w_{ij}} = \eta \sum_{\alpha} \frac{R_\alpha}{P_\alpha} \frac{\partial P_\alpha}{\partial w_{ij}}. \tag{14.12}$$

Combining equations (14.8), (14.9), and (14.10) we obtain the expression

$$\begin{aligned}\frac{\partial P_\alpha}{\partial w_{ij}} &= \frac{\gamma \sum_\beta \exp(-\gamma E_{\alpha\beta}) x_i^{\alpha\beta} x_j^{\alpha\beta}}{Z} \\ &\quad - \frac{\gamma \left(\sum_\beta \exp(-\gamma E_{\alpha\beta})\right) \sum_{\lambda\mu} \exp(-\gamma E_{\lambda\mu}) x_i^{\lambda\mu} x_j^{\lambda\mu}}{Z^2} \\ &= \gamma \left(\sum_\beta x_i^{\alpha\beta} x_j^{\alpha\beta} P_{\alpha\beta} - P_\alpha \langle x_i x_j \rangle_{free} \right).\end{aligned}$$

The expression $\langle x_i x_j \rangle_{free}$ denotes the expected value of the product of the unit states x_i and x_j in the free running network. Combining this result with (14.12) we get

$$\Delta w_{ij} = \eta\gamma \left(\sum_\alpha \frac{R_\alpha}{P_\alpha} \sum_\beta x_i^{\alpha\beta} x_j^{\alpha\beta} P_{\alpha\beta} - \sum_\alpha R_\alpha \langle x_i x_j \rangle_{free} \right). \tag{14.13}$$

We introduce the conditional probability $P_{\beta|\alpha}$ which is the probability that the hidden units assume state β when the input units are in state α, that is,

$$P_{\alpha\beta} = P_{\beta|\alpha} P_\alpha.$$

The expected value of the product of x_i and x_j during training (i.e., with clamped input) is given by

$$\langle x_i x_j \rangle_{fixed} = \sum_\alpha R_\alpha P_{\beta|\alpha} x_i^{\alpha\beta} x_j^{\alpha\beta}.$$

Using both definitions of the expected products and substituting in (14.13) leads to the expression

$$\begin{aligned} \Delta w_{ij} &= \eta\gamma \left(\sum_{\alpha,\beta} R_\alpha P_{\beta|\alpha} x_i^{\alpha\beta} x_j^{\alpha\beta} - \sum_\alpha R_\alpha \langle x_i x_j \rangle_{free} \right) \\ &= \eta\gamma \left(\langle x_i x_j \rangle_{fixed} - \langle x_i x_j \rangle_{free} \right) \end{aligned}$$

The above equation describes the learning method for a Boltzmann machine: the network is let run with clamped inputs and the average $\langle x_i x_j \rangle_{fixed}$ is computed. In a second pass, the network is let run without inputs. The average $\langle x_i x_j \rangle_{free}$ is computed and subtracted from the previously computed average. This provides us with an estimate of Δw_{ij}. The network weights are corrected until gradient descent has found a local minimum of the error measure D.

Note that Boltzmann learning resembles Hebbian learning. The values of $\langle x_i x_j \rangle_{fixed}$ and $\langle x_i x_j \rangle_{free}$ correspond to the entries of the stochastic correlation matrix of network states. The second term is subtracted and is interpreted as Hebbian "forgetting". This controlled loss of memory should prevent the network from learning false, spontaneously generated states. Obviously, Boltzmann learning in this form is computationally expensive for serial computers, since several passes over the whole data set and many update steps are needed.

14.3.2 Combinatorial optimization

Boltzmann machines can be used in those cases in which Hopfield networks become trapped in shallow local minima of the energy function. The energy function should have basins of attraction with clearly defined borders. Only in that case can we give a statistical guarantee that the global minimum will be reached. For problems such as the TSP this is not guaranteed. In these problems there are few global minima surrounded by many relatively deep local minima. The basins of attraction of the global minima are not much larger than the basins of attraction of the local minima. A Boltzmann machine can hardly improve on the results of the Hopfield network when the error function has an unfavorable shape. Boltzmann machines are therefore more important from a theoretical than from an engineering point of view. They can be used to model biological phenomena, such as conditioning and

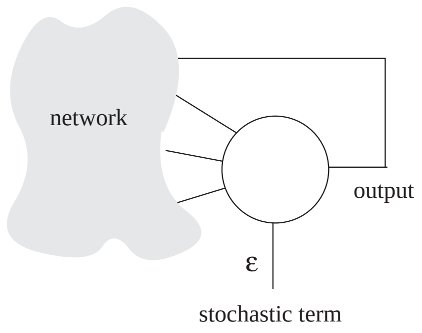


Fig. 14.5. Working principle of a Gauss machine

the emergence of memory. Biological neurons are stochastic systems and our models of artificial neural networks should reflect this fact.

Another possible realization of a stochastic Hopfield network consists in providing each unit in the network with a stochastic external input. Each unit i computes its net excitation $\sum_{j=1}^{N} w_{ij}x_j - \theta_i$ plus a stochastic term ε, as shown in Figure 14.5. The network dynamics is the standard Hopfield dynamics. Due to the stochastic term, some units update their states and can occasionally bring the network to states with a higher energy. Such stochastic systems have been called *Gauss machines*, because the stochastic term obeys a Gauss distribution. Some authors have compared Boltzmann and Gauss machines and find the latter more suitable for some tasks [13].

14.4 Historical and bibliographical remarks

Simulated annealing has been used for many years in the field of numerical optimization. The technique is a special case of the Monte Carlo method. Simulated annealing was brought into the mainstream of computing by the article published by Kirkpatrick, Gelatt, and Vecchi in 1983 [246]. The necessary conditions for convergence to a global minimum of a given function were studied in the classical paper by Geman and Geman [158]. An extensive analysis of the problem can be found in [1].

The introduction of stochastic computing units was proposed by several authors in different contexts. The Boltzmann machine concept was developed by Hinton and Sejnowski in the 1980s [191]. Their learning algorithm was the first proposed for stochastic networks and allowed dealing with hidden units in networks of the Hopfield type. In the following years other types of stochastic behavior were introduced. This led to models such as the *Cauchy machine* [423] and the Gauss machines mentioned already.

The role of noise in Hopfield networks has been studied by many physicists [25]. It can be shown, for example, that the number of false local minima that arise in Hopfield networks can be compensated with a stochastic dynamic, so that the statistical capacity of the network is improved.

Exercises

1. Solve the eight queens problem using a Boltzmann machine. Define the network's weights by hand.
2. Find with the computer the equilibrium distribution for a 10-node Boltzmann machine with randomly generated weights. Compute the Boltzmann distribution of the network and compare with the previous result.
3. Train a Boltzmann machine to generate solutions of the eight queens problem. Compare with your manually defined network of Exercise 1.

15

Kohonen Networks

15.1 Self-organization

In this chapter we consider *self-organizing networks*. The main difference between them and conventional models is that the correct output cannot be defined *a priori*, and therefore a numerical measure of the magnitude of the mapping error cannot be used. However, the learning process leads, as before, to the determination of well-defined network parameters for a given application.

15.1.1 Charting input space

In Chap. 5 we considered networks with a single layer which learn to identify clusters of vectors. This is also an example of a self-organizing system, since the correct output was not predefined and the mapping of weight vectors to cluster centroids is an automatic process.

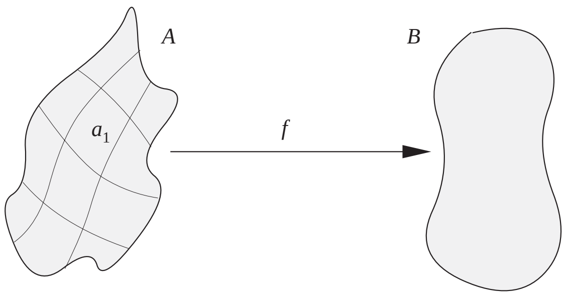


Fig. 15.1. A function $f : A \to B$

When a self-organizing network is used, an input vector is presented at each step. These vectors constitute the "environment" of the network. Each new input produces an adaptation of the parameters. If such modifications are correctly controlled, the network can build a kind of internal representation of the environment. Since in these networks learning and "production" phases can be overlapped, the representation can be updated continuously.

The best-known and most popular model of self-organizing networks is the topology-preserving map proposed by Teuvo Kohonen [254, 255]. So-called *Kohonen networks* are an embodiment of some of the ideas developed by Rosenblatt, von der Malsburg, and other researchers. If an input space is to be processed by a neural network, the first issue of importance is the structure of this space. A neural network with real inputs computes a function f defined from an input space A to an output space B. The region where f is defined can be covered by a Kohonen network in such a way that when, for example, an input vector is selected from the region a_1 shown in Figure 15.1, only one unit in the network fires. Such a tiling in which input space is classified in subregions is also called a chart or map of input space. Kohonen networks learn to create maps of the input space in a self-organizing way.

15.1.2 Topology preserving maps in the brain

Kohonen's model has a biological and mathematical background. It is well known in neurobiology that many structures in the brain have a linear or planar topology, that is, they extend in one or two dimensions. Sensory experience, on the other hand, is multidimensional. A simple event, such as the perception of color, presupposes interaction between three different kinds of light receptors. The eyes capture additional information about the structure, position, and texture of objects too. The question is: how do the planar structures in the brain manage to process such multidimensional signals? Put another way: how is the multidimensional input projected to the two-dimensional neuronal structures? This important question can be illustrated with two examples.

The visual cortex is a well-studied region in the posterior part of the human brain. Many neurons work together to decode and process visual impressions. It is interesting to know that the visual information is mapped as a two-dimensional projection on the cortex, despite the complexity of the pathway from the retina up to the cortical structures. Figure 15.2 shows a map of the visual cortex very similar to the one already discovered by Gordon Holmes at the beginning of the century [161]. The diagram on the right represents the whole visual field and its different regions. The inner circle represents the center of the visual field, whose contents are captured by the fovea. The diagram on the left uses three different shadings to show the correspondence between regions of the visual cortex and regions of the visual field. Two important phenomena can be observed: firstly, that neighboring regions of the visual field are processed by neighboring regions in the cortex; and secondly, that the surface of the visual cortex reserved for the processing of the information from the fovea is disproportionally large. This means that signals from the center of the visual field are processed in more detail and with higher resolution than signals from the periphery of the visual field. Visual acuity increases from the periphery to the center.

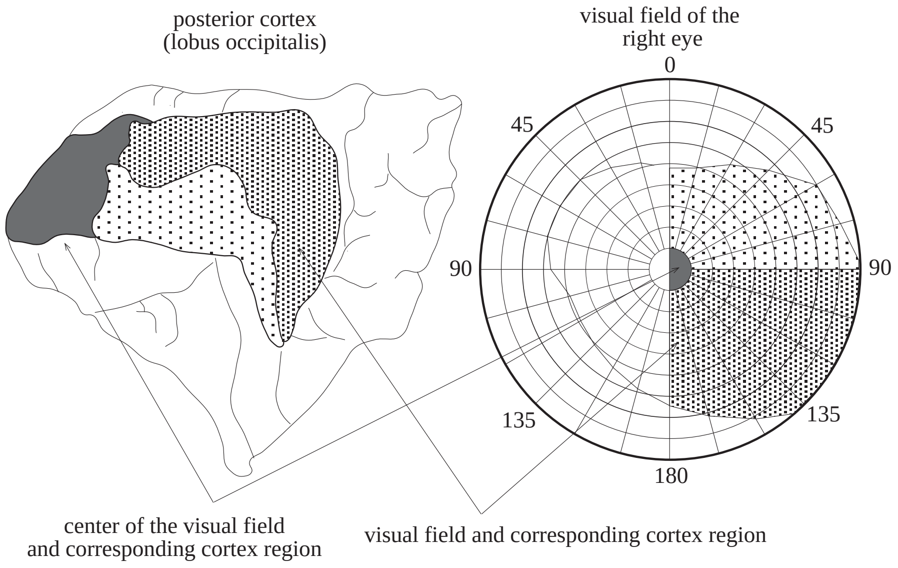

posterior cortex
(lobus occipitalis)
visual field of the
right eye
0
45
45
90
90
135
135
180
center of the visual field
and corresponding cortex region
visual field and corresponding cortex region

Fig. 15.2. Mapping of the visual field on the cortex

The visual cortex is therefore a kind of map of the visual field. Since this is a projection of the visual world onto the spherically arranged light receptors in the retina, a perfect correspondence between retina and cortex would need a spherical configuration of the latter. However, the center of the visual field maps to a proportionally larger region of the cortex. The form of the extended cortex should therefore resemble a deformed sphere. Physiological experiments have confirmed this conjecture [205].

In the human cortex we not only find a topologically ordered representation of the visual field but also of sensations coming from other organs. Figure 15.3 shows a slice of two regions of the brain: to the right the somatosensory cortex, responsible for processing mechanical inputs, and to the left the motor cortex, which controls the voluntary movement of different body parts. Both regions are present in each brain hemisphere and are located contiguous to each other.

Figure 15.3 shows that the areas of the brain responsible for processing body signals are distributed in a topology-preserving way. The region in charge of signals from the arms, for example, is located near to the region responsible for the hand. As can be seen, the spatial relations between the body parts are preserved as much as possible in the sensory cortex. The same phenomenon can be observed in the motor cortex.

Fig. 15.3. The somatosensory and motor cortex

Of course, all details of how the cortex processes sensory signals have not yet been elucidated. However, it seems a safe assumption that the first representation of the world built by the brain is a topological one, in which the exterior spatial relations are mapped to similar spatial relations in the cortex. One might think that the mapping of the sensory world onto brains is genetically determined, but experiments with cats that are blind in one eye have shown that those areas of the cortex which in a normal cat process information from the lost eye readapt and can process information from the other eye. This can only happen if the cat loses vision from one eye when the brain is still developing, that is, when it still possesses enough plasticity. This means that the topological correspondence between retina and cortex is not totally genetically determined and that sensory experience is necessary for the development of the correct neural circuitry [124].

Kohonen's model of self-organizing networks goes to the heart of this issue. His model works with elements not very different from the ones used by other researchers. More relevant is the definition of the neighborhood of a computing unit. Kohonen's networks are arrangements of computing nodes in one-, two-, or multi-dimensional lattices. The units have lateral connections to several neighbors. Examples of this kind of lateral coupling are the inhibitory connections used by von der Malsburg in his self-organizing models [284]. Connections of this type are also present, for example, in the human retina, as we discussed in Chap. 3.

15.2 Kohonen's model

In this section we deal with some examples of the ordered structures known as Kohonen networks. The grid of computing elements allows us to identify the immediate neighbors of a unit. This is very important, since during learning the weights of computing units and their neighbors are updated. The objective of such a learning approach is that neighboring units learn to react to closely related signals.

15.2.1 Learning algorithm

Consider the problem of charting an n-dimensional space using a one-dimensional chain of Kohonen units. The units are all arranged in sequence and are numbered from 1 to m (Figure 15.4). Each unit becomes the n-dimensional input $\mathbf{x}$ and computes the corresponding excitation. The n-dimensional weight vectors $\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_m$ are used for the computation. The objective of the charting process is that each unit learns to specialize on different regions of input space. When an input from such a region is fed into the network, the corresponding unit should compute the maximum excitation. Kohonen's learning algorithm is used to guarantee that this effect is achieved.

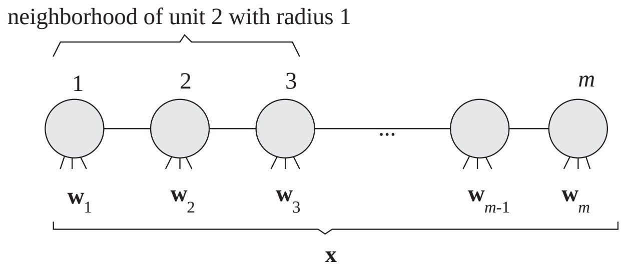


Fig. 15.4. A one-dimensional lattice of computing units

A Kohonen unit computes the Euclidian distance between an input $\mathbf{x}$ and its weight vector $\mathbf{w}$. This new definition of excitation is more appropriate for certain applications and also easier to visualize. In the Kohonen one-dimensional network, the neighborhood of radius 1 of a unit at the k-th position consists of the units at the positions $k-1$ and $k+1$. Units at both ends of the chain have asymmetrical neighborhoods. The neighborhood of radius r of unit k consists of all units located up to r positions from k to the left or to the right of the chain.

Kohonen learning uses a neighborhood function ϕ, whose value $\phi(i,k)$ represents the strength of the coupling between unit i and unit k during the training process. A simple choice is defining $\phi(i,k) = 1$ for all units i in a neighborhood of radius r of unit k and $\phi(i,k) = 0$ for all other units. We will later discuss the problems that can arise when some kinds of neighborhood functions are chosen. The learning algorithm for Kohonen networks is the following:

Algorithm 15.2.1 *Kohonen learning*

start: The n-dimensional weight vectors $\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_m$ of the m computing units are selected at random. An initial radius r, a learning constant η, and a neighborhood function ϕ are selected.

step 1: Select an input vector ξ using the desired probability distribution over the input space.

step 2: The unit k with the maximum excitation is selected (that is, for which the distance between $\mathbf{w}_i$ and ξ is minimal, $i = 1, \ldots, m$).

step 3: The weight vectors are updated using the neighborhood function and the update rule

$$\mathbf{w}_i \leftarrow \mathbf{w}_i + \eta\phi(i,k)(\xi - \mathbf{w}_i), \quad \text{for} \quad i = 1, \ldots, m.$$

step 4: Stop if the maximum number of iterations has been reached; otherwise modify η and ϕ as scheduled and continue with step 1.

The modifications of the weight vectors (step 3) attracts them in the direction of the input ξ. By repeating this simple process several times, we expect to arrive at a uniform distribution of weight vectors in input space (if the inputs have also been uniformly selected). The radius of the neighborhood is reduced according to a previous plan, which we call a *schedule*. The effect is that each time a unit is updated, neighboring units are also updated. If the weight vector of a unit is attracted to a region in input space, the neighbors are also attracted, although to a lesser degree. During the learning process both the size of the neighborhood and the value of ϕ fall gradually, so that the influence of each unit upon its neighbors is reduced. The learning constant controls the magnitude of the weight updates and is also reduced gradually. The net effect of the selected schedule is to produce larger corrections at the beginning of training than at the end.

Figure 15.5 shows the results of an experiment with a one-dimensional Kohonen network. Each unit is represented by a dot. The input domain is a triangle. At the end of the learning process the weight vectors reach a distribution which transforms each unit into a "representative" of a small region of input space. The unit in the lower corner, for example, is the one

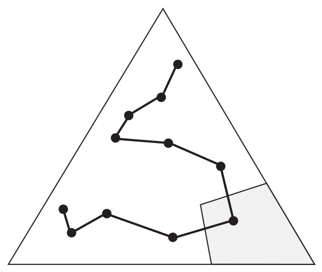


Fig. 15.5. Map of a triangular region

which responds with the largest excitation for vectors in the shaded region. If we adopt a "winner-takes-all" activation strategy, then it will be the only one to fire.

The same experiment can be repeated for differently shaped domains. The chain of Kohonen units will adopt the form of a so-called *Peano curve*. Figure 15.6 is a series of snapshots of the learning process from 0 to 25000 iterations [255]. At the beginning, before training starts, the chain is distributed randomly in the domain of definition. The chain unwraps little by little and the units distribute gradually in input space. Finally, when learning approaches its end, only small corrections affect the unit's weights. At that point, the neighborhood radius has been reduced to zero and the learning constant has reached a small value.

Fig. 15.6. Mapping a chain to a triangle

Kohonen networks can be arranged in multidimensional grids. An interesting choice is a planar network, as shown in Figure 15.7. The neighborhood of radius r of unit k consists, in this case, of all other units located at most r places to the left or right, up or down in the grid. With this convention, the neighborhood of a unit is a quadratic portion of the network. Of course we can define more sophisticated neighborhoods, but this simple approach is all that is needed in most applications.

Figure 15.7 shows the flattening of a two-dimensional Kohonen network in a quadratic input space. The four diagrams display the state of the network after 100, 1000, 5000, and 10000 iterations. In the second diagram several iterations have been overlapped to give a feeling of the iteration process. Since in this experiment the dimension of the input domain and of the network are the same, the learning process reaches a very satisfactory result.

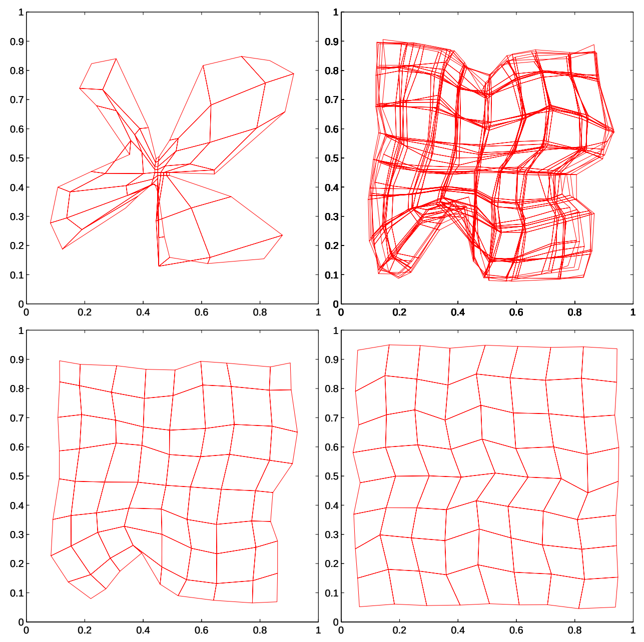


Fig. 15.7. Mapping a square with a two-dimensional lattice. The diagram on the upper right shows some overlapped iterations of the learning process. The diagram below it is the final state after 10000 iterations.

Settling on a stable state is not so easy in the case of multidimensional networks. There are many factors which play a role in the convergence process, such as the size of the selected neighborhood, the shape of the neighborhood function and the scheduling selected to modify both. Figure 15.8 shows an example of a network which has reached a state very difficult to correct. A knot has appeared during the training process and, if the plasticity of the network has reached a low level, the knot will not be undone by further training, as the overlapped iterations in the diagram on the right, in Figure 15.8 show.

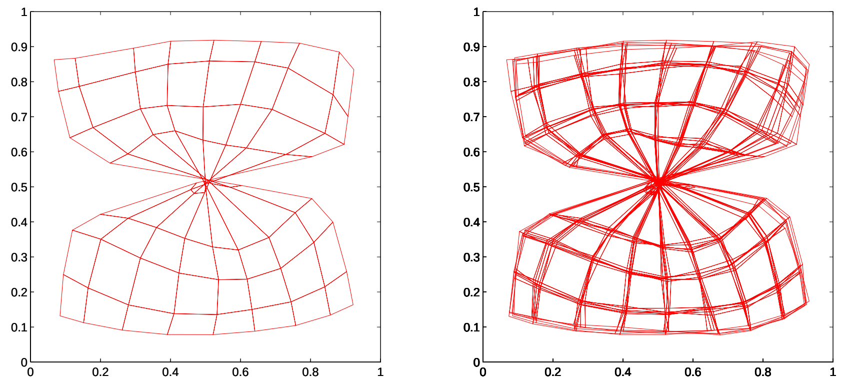


Fig. 15.8. Planar network with a knot

Several proofs of convergence have been given for one-dimensional Kohonen networks in one-dimensional domains. There is no general proof of convergence for multidimensional networks.

15.2.2 Mapping high-dimensional spaces

Usually, when an empirical data set is selected, we do not know its real dimension. Even if the input vectors are of dimension n, it could be that the data concentrates on a manifold of lower dimension. In general it is not obvious which network dimension should be used for a given data set. This general problem led Kohonen to consider what happens when a low-dimensional network is used to map a higher-dimensional space. In this case the network must fold in order to fill the available space. Figure 15.9 shows, in the middle, the result of an experiment in which a two-dimensional network was used to chart a three-dimensional box. As can be seen, the network extends in the x and y dimensions and folds in the z direction. The units in the network try as hard as possible to fill the available space, but their quadratic neighborhood poses some limits to this process. Figure 15.9 shows, on the left and on the right, which portions of the network approach the upper or the lower side of the box. The black and white stripes resemble a zebra pattern.

Remember that earlier we discussed the problem of adapting the planar brain cortex to a multidimensional sensory world. There are some indications that a self-organizing process of the kind shown in Figure 15.9 could also be taking place in the human brain, although, of course, much of the brain structure emerges pre-wired at birth. The experiments show that the foldings of the planar network lead to stripes of alternating colors, that is, stripes which

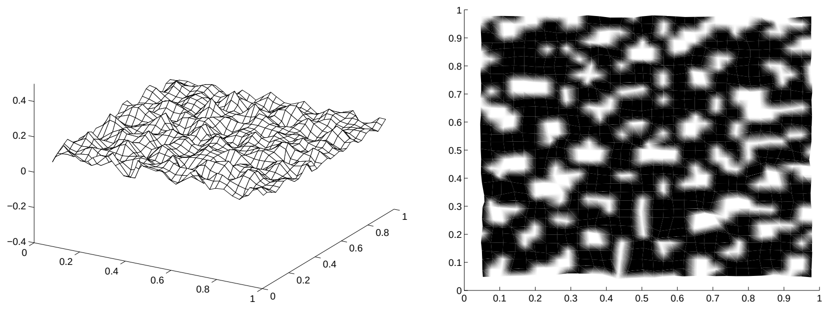


Fig. 15.9. Two-dimensional map of a three-dimensional region

map alternately to one side or the other of input space (for the z dimension). A commonly cited example for this kind of structure in the human brain is the visual cortex. The brain actually processes not one but two visual images, one displaced with respect to the other. In this case the input domain consists of two planar regions (the two sides of the box of Figure 15.9). The planar cortex must fold in the same way in order to respond optimally to input from one or other side of the input domain. The result is the appearance of the stripes of ocular dominance studied by neurobiologists in recent years. Figure 15.10 shows a representation of the ocular dominance columns in LeVays' reconstruction [205]. It is interesting to compare these stripes with the ones found in our simple experiment with the Kohonen network.

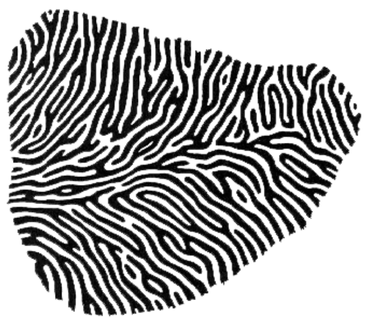


Fig. 15.10. Diagram of eye dominance in the visual cortex. Black stripes represent one eye, white stripes the other.

These modest examples show the kinds of interesting consequence that can be derived from Kohonen's model. In principle Kohonen networks resemble the unsupervised networks we discussed in Chap. 5. Here and there we try to chart an input space distributing computing units to define a vector quantization. The main difference is that Kohonen networks have a *predefined topology.* They are organized in a grid and the learning problem is to find a way to distribute this grid in input space. One could follow another approach, such as using a k-means type of algorithm to distribute $n \times n$ units in a quadratic domain. We could thereafter use these units as the vertices of a planar grid, which we define at will. This is certainly better if we are only interested in obtaining a grid for our quadratic domain. If we are interested in understanding how a given grid (i.e., the cortex) adapts to a complex domain, we have to start training with the folded grid and terminate training with an unfolded grid. In this case what we are investigating is the interaction between "nurture and nature". What initial configurations lead to convergence? How is convergence time affected by a preordered network compared to a fully random network? All these issues are biologically relevant and can only be answered using the full Kohonen model or its equivalent.

15.3 Analysis of convergence

We now turn to the difficult question of analyzing the convergence of Kohonen's learning algorithm. We will not go into detail in this respect, since the necessary mathematical machinery is rather complex and would take too much space in this chapter. We will content ourselves with looking at the conditions for stability when the network has arrived at an ordered state, and give some useful references for more detailed analysis.

15.3.1 Potential function – the one-dimensional case

Consider the simplest Kohonen network – it consists of just one unit and the input domain is the one-dimensional interval $[a, b]$. The learning algorithm will lead to convergence of the sole weight x in the middle of the interval $[a, b]$.

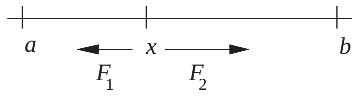


Fig. 15.11. The weight x in the interval $[a, b]$

In our example Kohonen learning consists of the update rule

$$x_n = x_{n-1} + \eta(\xi - x_{n-1}),$$

where x_n and x_{n-1} represent the values of the unit's weight in steps n and $n-1$ and ξ is a random number in the interval $[a, b]$. If $0 < \eta \leq 1$, the series $x_1, x_2, \ldots$ cannot leave the interval $[a, b]$, that is, it is bounded. Since the expected value $\langle x \rangle$ of x is also bounded, this means that the expected value of the derivative of x with respect to t is zero,

$$\left\langle \frac{dx}{dt} \right\rangle = 0,$$

otherwise the expected value of x will eventually be lower than a or greater than b. Since

$$\left\langle \frac{dx}{dt} \right\rangle = \eta\left(\langle\xi\rangle - \langle x\rangle\right) = \eta\left(\frac{a+b}{2} - \langle x\rangle\right),$$

this means that

$$\langle x\rangle = \frac{a+b}{2}.$$

The same technique can be used to analyze the stable states of the general one-dimensional case. Let n units be arranged in a one-dimensional chain whose respective weights are denoted by $x^1, x^2, \ldots, x^n$. The network is used to chart the one-dimensional interval $[a,b]$. Assume that the weights are arranged monotonically and in ascending order, i.e. $a < x^1 < x^2 < \cdots < x^n < b$. We want to show that the expected values of the weights are given by

$$\left\langle x^i \right\rangle = a + (2i-1)\frac{b-a}{2n}. \qquad (15.1)$$

All network weights are distributed as shown in Figure 15.12.

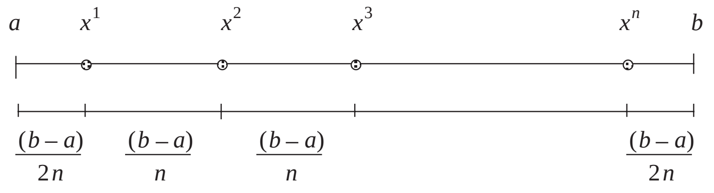


Fig. 15.12. Distribution of the network weights in the interval $[a,b]$

The distribution is statistically stable for a one-dimensional Kohonen network because the attraction on each weight of its domain is zero on average. The actual weights oscillate around the expected values $\left\langle x^1\right\rangle, \ldots, \langle x^n\rangle$. Since, in Kohonen learning, the weights stay bounded, it holds for $i = 1, 2, \ldots, n$ that

$$\left\langle \frac{dx^i}{dt} \right\rangle = 0 \qquad (15.2)$$

The learning algorithm does not modify the relative positions of the weights. Equation (15.2) holds if the expected values $\left\langle x^1\right\rangle, \ldots, \langle x^n\rangle$ are distributed homogeneously, that is, in such a way that attraction from the right balances the attraction from the left. This is only possible with the distribution given by (15.1). Note that we did not make any assumptions on the form of the neighborhood function (we effectively set it to zero). If the neighborhood function is taken into account, a small difference arises at both ends of the chain, because the attraction from one side of the neighborhood is not balanced by the corresponding attraction from the other. This can be observed during training of Kohonen networks, for example in the lower right diagram in Figure 15.7.

15.3.2 The two-dimensional case

Stable states for the two-dimensional case can be analyzed in a similar way. Assume that a two-dimensional Kohonen network with $n \times n$ units is used to map the interval $[a, b] \times [c, d]$. Each unit has four immediate neighbors, with the exception of the units at the borders (Figure 15.13).

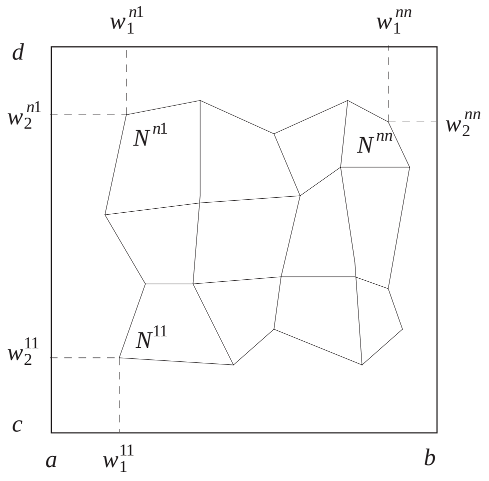


Fig. 15.13. Two-dimensional map

Denote the unit in the lower left corner by N^{11} and the unit in the upper right corner by N^{nn}. Unit N^{ij} is located at row i and column j of the network. Consider a monotonic ordering of the network weights, i.e.,

$$w_1^{ij} < w_1^{ik} \qquad \text{if } \; j < k \tag{15.3}$$

and

$$w_2^{ij} < w_2^{kj} \qquad \text{if } \; i < k, \tag{15.4}$$

where w_1^{ij} and w_2^{ij} denote the two weights of unit N^{ij}.

Kohonen learning is started with an ordered configuration of this type. It is intuitively clear that the weights will distribute homogeneously in the input domain. The two-dimensional problem can be broken down into two one-dimensional problems. Let

$$w_1^j = \frac{1}{n}\sum_{i=1}^{n} w_1^{ij}$$

denote the average value of the weights of all units in the j-th column. Since equation (15.4) holds, these average values are monotonically arranged, that is, for the first coordinate we have

$$a < w_1^1 < w_1^2 < \cdots < w_1^n < b.$$

If the average values are monotonically distributed, their expected values $\langle w_1^i \rangle$ will reach a homogeneous distribution in the interval $[a, b]$. Assuming that the neighborhood function has been set to zero, the units' weights $w_1^{11}, w_1^{21}, \ldots, w_1^{n1}$ of the first column will oscillate around the average value $\langle w_1^1 \rangle$. The same considerations can be made for the average values of the weights of each row of units. If the learning constant is small enough, the initial distribution will converge to a stable state.

Unfortunately, to arrive at this ideal state it is first necessary to unfold the randomly initialized Kohonen network. As Figure 15.8 shows, this process can fail at an early stage. The question is, therefore, under which conditions can such a planar network arrive at the unfolded state. Many theoreticians have tried to give an answer, but a general solution remains to be found [66, 91, 367].

15.3.3 Effect of a unit's neighborhood

In the previous section we did not consider the effect of a unit's neighborhood on its final stable state. Assume that the neighborhood function of a unit is given by

$$\phi(i,k) = \begin{cases} 1 & \text{if } i = k \\ 1/2 & \text{if } i = k \pm 1 \\ 0 & \text{otherwise} \end{cases}$$

Let a linear Kohonen network be trained with this neighborhood function. The stable ordering that is eventually reached (Figure 15.14) does not divide the input domain in equal segments. The stable mean values for x_1 and x_2 are $-1/4$ and $1/4$ respectively (see Exercise 2).

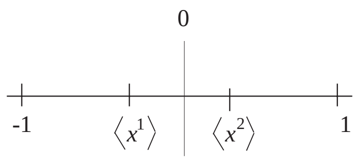


Fig. 15.14. Two weights in the interval $[-1, 1]$

The neighborhood function produces a concentration of units around the center of the distribution. The exact pattern depends on the neighborhood function and the neighborhood radius. Strong coupling of the grid's units attracts them to its center. The correct learning strategy is therefore to start training with a strong coupling, which is reduced gradually as learning progresses.

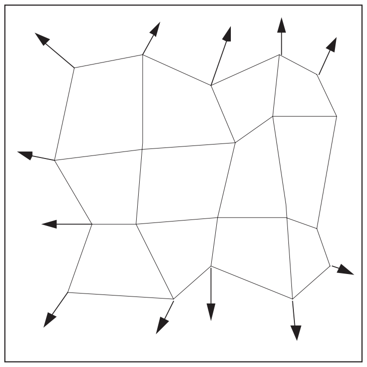


Fig. 15.15. Two-dimensional lattice being straightened out

In the case of a two-dimensional network (Figure 15.15), this training strategy concentrates the network towards the center. However, the periphery of the distribution attracts the network, unfolds it, and helps to achieve convergence. A way to help this process is to initialize the network with small weights (when the empirical data is centered at the origin).

15.3.4 Metastable states

In 1986 Cotrell and Fort showed that one-dimensional Kohonen networks converge to an ordered state if the input is selected from a uniform distribution [91]. Later Bouton and Pages extended this result for other distributions [66]. This has sometimes been interpreted as meaning that one-dimensional Kohonen networks always converge to a stable ordered state. However, this is only true for the kind of learning rule used in the proof of the convergence results. For a chain of weights $w_1, w_2, \ldots, w_n$, the weight updates are given by

$$w_k = w_k + \gamma(\xi - w_k),$$

where k is equal to the index of the nearest weight to the input ξ and each of its two neighbors (with the obligatory exceptions at both ends) and γ is a learning constant. Note that this learning rule implies that the neighborhood function does not decay from one position in the chain to the next. Usually, however, a neighborhood function $\phi(i,k)$ is used to train the network and the question arises whether under some circumstances the one-dimensional Kohonen network could possibly converge to a disordered state. This is indeed the case when the neighborhood function decays too fast.

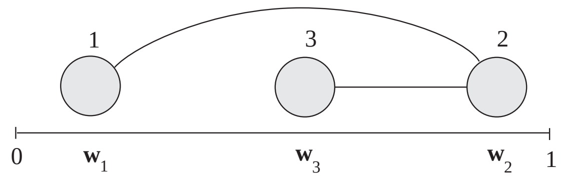


Fig. 15.16. A disordered one-dimensional network

To see how these *metastable* states arise, consider a simple chain of three units, which is started in the disordered combination in the interval $[0,1]$, as shown in Figure 15.16. The chain will be in equilibrium if the expected value of the total attraction on each weight is zero. Consider a simple neighborhood function

$$\phi(i,k) = \begin{cases} 1 \text{ if } i = k \\ a \text{ if } i = k \pm 1 \\ 0 \text{ otherwise} \end{cases}$$

and the learning rule

$$w_k = w_k + \gamma\phi(i,k)(\xi - w_k),$$

where all variables have the same meaning as explained before and a is a real positive constant. We will denote the total attractive "force" on each weight by f_1, f_2, f_3. In the configuration shown in Figure 15.16 the total attractions are (see Exercise 4):

$$f_1 = (-\frac{3}{4} - a)w_1 + \frac{a}{4}w_2 + (\frac{1}{4} + \frac{a}{4})w_3 + \frac{a}{2} \tag{15.5}$$

$$f_2 = \frac{a}{4}w_1 + (-\frac{3}{4} - a)w_2 + (\frac{1}{4} + \frac{a}{4})w_3 + \frac{1}{2} \tag{15.6}$$

$$f_3 = \frac{1}{4}w_1 + (\frac{1}{4} + \frac{a}{4})w_2 + (-\frac{1}{2} - \frac{3a}{4})w_3 + \frac{a}{2} \tag{15.7}$$

This is a system of linear equations for the three weights w_1, w_2 and w_3. The associated "force" matrix is:

$$F = \frac{1}{4}\begin{pmatrix} -a-3 & a & 1+a \\ a & -a-3 & a+1 \\ 1 & a+1 & -3a-2 \end{pmatrix}$$

The matrix F can be considered the negative Hesse matrix of a potential function $U(w_1, w_2, w_3)$. The solution is stable if the matrix $-F$ is positive definite. The solution of the system when $a = 0$, that is, when Kohonen units do not attract their neighbors, is $w_1 = 1/6, w_2 = 5/6, w_3 = 1/2$. This is a *stable disordered state*. If a is increased slowly, the matrix of the system of linear equations is still invertible and the solutions comply with the constraints of the problem (each weight lies in the interval $[0,1]$). For $a = 0.01$, for example, the solutions $w_1 = 0.182989, w_2 = 0.838618, w_3 = 0.517238$ can be found numerically. This is also a stable disordered state as can be proved (Exercise 4) by computing the eigenvalues of the matrix $-F$, which are given by

$$\begin{aligned}
\lambda_1 &= \frac{1}{3+2a} \\
\lambda_2 &= \frac{5+3a+\sqrt{(5+3a)^2-4(4+6a-a^2)}}{2(4+6a-a^2)} \\
\lambda_3 &= \frac{5+3a-\sqrt{(5+3a)^2-4(4+6a-a^2)}}{2(4+6a-a^2)}.
\end{aligned}$$

For small values of a all three eigenvalues are real and positive. This means that $-F$ is positive definite.

We can also numerically compute the maximum value of a for which it is still possible to find metastable states. For $a = 0.3$ the stable solutions no longer satisfy the constraints of the problem (w_2 becomes larger than 1). We should expect convergence to an *ordered* state for values of a above this threshold.

The example of one-dimensional chains illustrates some of the problems associated with guaranteeing convergence of Kohonen networks. In the multidimensional case care must be taken to ensure that the appropriate neighorhood is chosen, that a good initialization is used and that the cooling scheduling does not freeze the network too soon. And still another problem remains: that of choosing the dimension of the model.

15.3.5 What dimension for Kohonen networks?

In many cases we have experimental data which is coded using n real values, but whose effective dimension is much lower. Consider the case in which the data set consists of the points in the surface of a sphere in three-dimensional space. Although the input vectors have three components, a two-dimensional Kohonen network will do a better job of charting this input space than a three-dimensional one. Some researchers have proposed computing the effective dimension of the data before selecting the dimension of the Kohonen network, since this can later provide a smoother approximation to the data.

The dimension of the data set can be computed experimentally by measuring the variation in the number of data points closer to another data point than a given ε, when ε is gradually increased or decreased. If, for example, the data set lies on a plane in three-dimensional space and we select a data point ξ randomly, we can plot the number of data points $N(\varepsilon)$ not further away from ξ than ε. This number should follow the power law $N(\varepsilon) \approx \varepsilon^2$. If this is the case, we settle for a two-dimensional network. Normally, the way to make this computation is to draw a plot of $\log(N(\varepsilon)$ against $\log(\varepsilon)$. The slope of the regression curve as ε goes to zero is the fractal dimension of the data set.

Finding the appropriate power law means that in some cases a function has to be fitted to the measurements. Since the dimension of the data can sometimes best be approximated by a fraction, we speak of the *fractal dimension* of the data. It has been shown experimentally that if the dimension of the Kohonen network approaches the fractal dimension of the data, the interpolation error is smaller than for other network sizes [410]. This can be understood as meaning that the units in the network are used optimally to approximate input space. Other methods of measuring the fractal dimension of data are discussed by Barnsley [42].

15.4 Applications

Kohonen networks can adapt to domains with the most exotic structures and have been applied in many different fields, as will be shown in the following sections.

15.4.1 Approximation of functions

A Kohonen network can be used to approximate the continuous real function f in the domain of definition $[0,1] \times [0,1]$. The set $P = \{(x, y, f(x, y)) | x, y \in [0,1]\}$ is a surface in three-dimensional space. We would like to adapt a planar grid to this surface. In this case the set P is the domain which we try to map with the Kohonen network.

After the learning algorithm is started, the planar network moves in the direction of P and distributes itself to cover the domain. Figure 15.17 shows the result of an experiment in which the function $z = 5 \sin x + y$ had to be learned. The combinations of x and y were generated in a small domain. At the end of the training process the network has "learned" the function f in the region of interest.

The function used in the example above is the one which guarantees optimal control of a *pole balancing system*. Such a system consists of a pole (of mass 1) attached to a moving car. The pole can rotate at the point of attachment and the car can only move to the left or to the right. The pole should be kept in equilibrium by moving the car in one direction or the other. The necessary force to keep the pole in equilibrium is given by $f(\theta) = \alpha \sin \theta + \beta \, d\theta/dt$ [368], where θ represents the angle between the pole and the vertical, and α and β are constants. (Figure 15.18). For small values of θ a linear approximation can be used. Since a Kohonen network can learn the function f, it can also be used to provide the automatic control for the pole balancing system.

When a combination of x and y is given (in this case θ and $d\theta/dt$), the unit (i, j) is found for which the Euclidian distance between its associated weights $w_1^{(i,j)}$ and $w_2^{(i,j)}$ and $(\theta, d\theta/dt)$ is minimal. The value of the function at this point is the learned $w_3^{(i,j)}$, that is, an approximation to the value of the

Fig. 15.17. Control surface of the balancing pole [Ritter et al. 1990]

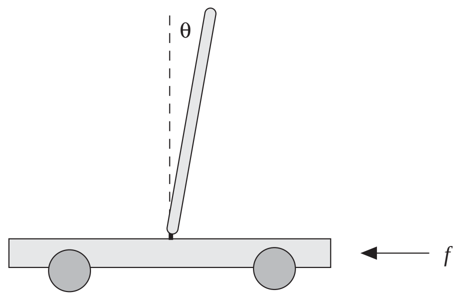


Fig. 15.18. A balancing pole

function f. The network is therefore a kind of look-up table of the values of f. The table can be made as sparse or as dense as needed for the application at hand. Using a table is in general more efficient than computing the function each time from scratch. If the function is not analytically given, but has been learned using some input-output examples, Kohonen networks resemble backpropagation networks. The Kohonen network can continue adapting and, in this way, if some parameters of the system change (because some parts begin to wear), such a modification is automatically taken into account. The Kohonen network behaves like an adaptive table, which can be built using a minimal amount of hardware. This has made Kohonen networks an interesting choice in robotic applications.

15.4.2 Inverse kinematics

A second application of Kohonen networks is mapping the configuration space of a mechanical arm using a two-dimensional grid. Assume that a robot arm with two joints, as shown in Figure 15.19, is used to reach to all points in a table. The robot arm has two degrees of freedom, i.e., the angles α and β. The network is trained using a feedback signal from the manipulator. The two-dimensional Kohonen network is distributed homogenously in the quadratic domain. Now the tip of the arm is positioned manually at randomly selected points of the working surface. The weights of the Kohonen unit which is nearest to this point are updated (and also the weights of the neighbors), but the inputs for the update are the joints' angles. The network therefore charts the space of degrees of freedom. The parameters stored as weights at each node represent the combination of angles needed to reach a certain point, as shown in Figure 15.19. In this case the Kohonen network can again be considered a parameter table. Figure 15.19 shows that if the robot arm must be positioned at the selected point, it is only necessary to read the parameters from the grid. Note that in this case we assumed that the learning process avoids reaching the same point with a different parameter combination.

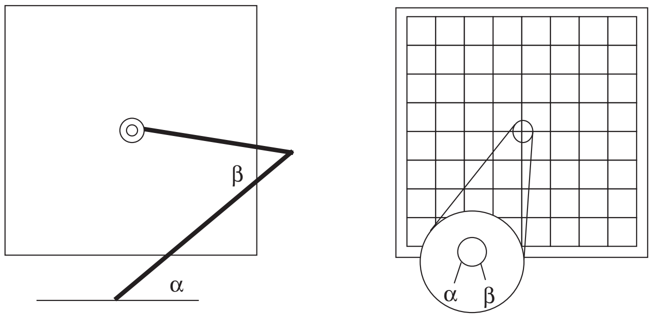


Fig. 15.19. Robot arm (left) and trained network (right)

More interesting is the case where we want to displace the tip of the manipulator from point A to point B. It is only necessary to find a path from A to B on the grid. All units in this path provide the necessary parameters for the movement. Positions in between can be interpolated from the data.

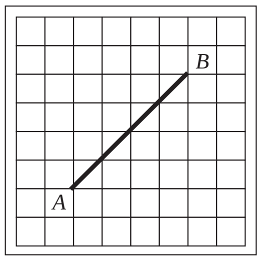


Fig. 15.20. Optimal path from A to B

The problem can be made more difficult if obstacles are allowed in the working surface. Figure 15.21 shows such a hurdle. If a Kohonen network is trained, it charts the configuration space of the manipulator in such a way as to avoid the forbidden zones. If we distribute the network in the working area according to the stored parameters, the result is the one shown to the left of Figure 15.21. The network itself is still a two-dimensional grid. If we want to move the manipulator from A to B, we can plan the movement on the grid in the way we discussed before. The actual path selection is the one shown on the right of Figure 15.21. The manipulator moves to avoid the obstacle.

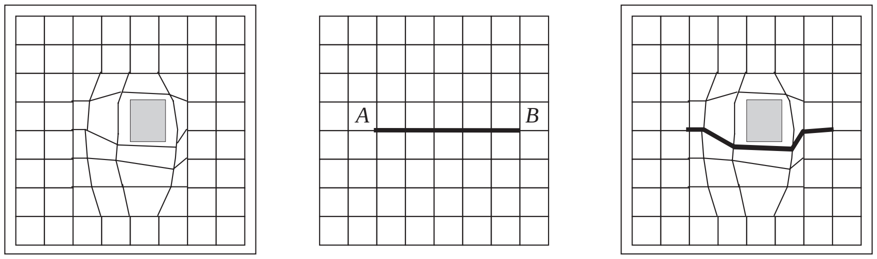


Fig. 15.21. Obstacles in the work area

This method can only be employed if we have previously verified that the edges of the network provide valid interpolations. It could be that two adjacent nodes contain valid information, but the path between them leads to a collision. Before using the network in the way discussed above, a previous validation step should guarantee that the edges of the grid are collision-free.

The Kohonen network can be used only if the obstacles do not change their positions. It is conceivable that if the changes occur very slowly, a fast learning algorithm could adapt the network to the changing circumstances. Kohonen networks behave in this kind of application as a generalized coordinate system. Finding the shortest path in the network corresponds to finding the shortest path in the domain of the problem, even when the movement performed by the manipulator is rather complex and nonlinear.

15.5 Historical and bibliographical remarks

Many applications have been developed since Kohonen first proposed the kind of networks which now bear his name. It is fascinating to see how such a simple model can offer plausible explanations for certain biological phenomena. Kohonen networks have even been used in the field of combinatorics, for example, to solve the Traveling Salesman Problem with the so-called *elastic net* algorithm [117]. A ring of units is used to encircle the cities to be visited in the Euclidian plane and Kohonen learning wraps this ring around them, so that a round trip of nearly minimal length is found.

The first applications in the field of robotics were developed and perfected in the 1980s. Kohonen has carried out some research on the application of topology-preserving networks in such diverse fields as speech recognition and structuring of semantic networks. The *phonetic typewriter* is a system based on a Kohonen map of the space of phonemes. A word can be recognized by observing the path in phoneme space reconstructed by the network [431].

Even if the original Kohonen model requires some non-biological assumptions (for example, the exchange of non-local information) there are ways to circumvent this problem. Its most important features are its self-organizing properties which arise from a very simple adaptation rule. Understanding the mathematics of such self-organizing processes is a very active field of research [222].

Exercises

1. Implement a Kohonen one-dimensional network and map a two-dimensional square. You should get a Peano curve of the type discussed in this chapter.
2. Prove that the neighborhood function of the linear network in Sect. 15.3.3 has a stable state at $x_1 = -1/4$ and $x_2 = 1/4$. Derive a general expression for a linear chain of n elements and the same neighborhood function.
3. Is it possible for a two-dimensional Kohonen network in an ordered state to become disordered after some iterations of the learning algorithm? Consider the case of a two-dimensional domain.
4. Derive the expected values of f_1, f_2 and f_3 given in equations (15.5), (15.6), and (15.7).
5. Give an example of an obstacle in the working field of the robot arm discussed in the text that makes an edge of the Kohonen network unusable, although the obstacle is far away from the edge.

16

Modular Neural Networks

In the previous chapters we have discussed different models of neural networks – linear, recurrent, supervised, unsupervised, self-organizing, etc. Each kind of network relies on a different theoretical or practical approach. In this chapter we investigate how those different models can be combined. We transform each single network in a module that can be freely intermixed with modules of other types. In this way we arrive at the concept of *modular neural networks*. Several general issues have led to the development of modular systems [153]:

- *Reducing model complexity.* As we discussed in Chap. 10 the way to reduce training time is to control the degrees of freedom of the system.
- *Incorporating knowledge.* Complete modules are an extension of the approach discussed in Sect. 10.3.5 of learning with hints.
- *Data fusion and prediction averaging.* Committees of networks can be considered as composite systems made of similar elements (Sect. 9.1.6)
- *Combination of techniques.* More than one method or class of network can be used as building block.
- *Learning different tasks simultaneously.* Trained modules may be shared among systems designed for different tasks.
- *Robustness and incrementality.* The combined network can grow gradually and can be made fault-tolerant.

Modular neural networks, as combined structures, have also a biological background: Natural neural systems are composed of a hierarchy of networks built of elements specialized for different tasks. In general, combined networks are more powerful than flat unstructured ones.

16.1 Constructive algorithms for modular networks

Before considering networks with a self-organizing layer, we review some techniques that have been proposed to provide structure to the hidden layer of feed-forward neural networks or to the complete system (as a kind of decision tree).

16.1.1 Cascade correlation

An important but difficult problem in neural network modeling, is the selection of the appropriate number of hidden units. The *cascade correlation algorithm*, proposed by Fahlman and Lebiere [131], addresses this issue by recruiting new units according to the residual approximation error. The algorithm succeeds in giving structure to the network and reducing the training time necessary for a given task [391].

Figure 16.1 shows a network trained and structured using cascade correlation. Assume for the moment that a single output unit is required. The algorithms starts with zero hidden units and adds one at a time according to the residual error. The diagram on the right in Figure 16.1 shows the start configuration. The output unit is trained to minimize the quadratic error. Training stops when the error has leveled off. If the average quadratic error is still greater than the desired upper bound, we must add a hidden unit and retrain the network.

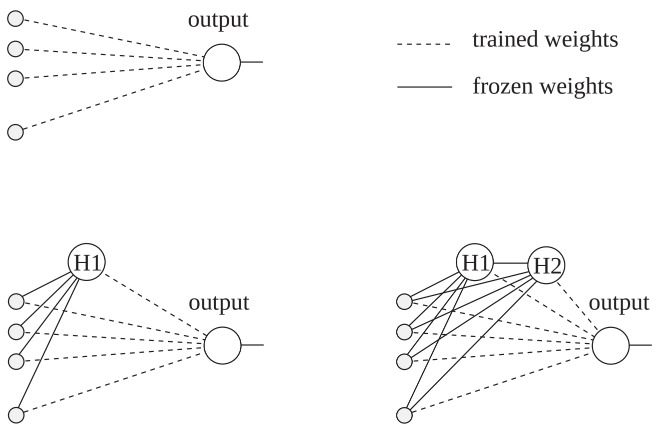


Fig. 16.1. The cascade correlation architecture

Denote the mean error of the network by $\bar{E}$ and the error for pattern $i = 1, 2, \ldots, p$, by E_i. A hidden unit is trained, isolated from the rest of the network, to *maximize* the absolute value of the correlation between $V_i - \bar{V}$ and $E_i - \bar{E}$, where V_i denotes the unit's output for the i-th pattern and $\bar{V}$ its average output. The quantity to be maximized is therefore

$$S = |\sum_{i=1}^{p}(V_i - \hat{V})(E_i - \hat{E})|. \tag{16.1}$$

The motivation behind this step is to specialize the hidden unit to the detection of the residual error of the network. The backpropagation algorithm is applied as usual, taking care to deal with the sign of the argument of the absolute value operator. One way of avoiding taking absolute values at all, is to train a weight at the output of the hidden unit, so that it assumes the appropriate sign (see Exercise 1).

Once the hidden unit H1 has been trained, i.e., when the correlation level cannot be further improved, the unit is added to the network as shown in Figure 16.1 (left diagram). The weights of the hidden unit are frozen. The output unit receives information now from the input sites and from the hidden unit H1. All the weights of the output unit are retrained until the error levels off and we test if a new hidden unit is necessary. Any new hidden unit receives an input from the input sites and from all other previously defined hidden units. The algorithm continues adding hidden units until we are satisfied with the approximation error.

The advantage of the algorithm, regarding learning time, is that in every iteration a single layer of weights has to be trained. Basically, we only train one sigmoidal unit at a time. The final network has more structure than the usual flat feed-forward networks and, if training proceeds correctly, we will stop when the minimum number of necessary hidden units has been selected. To guarantee this, several hidden units can be trained at each step and the one with the highest correlation selected for inclusion in the network [98].

In the case of more than one output unit, the average of the correlation of all output units is maximized at each step. A summation over all output units is introduced in equation (16.1).

16.1.2 Optimal modules and mixtures of experts

Cascade correlation can be considered a special case of a more general strategy that consists in training special modules adapted for one task. As long as they produce continuous and differentiable functions, it is possible to use them in any backpropagation network. The learning algorithm is applied as usual, with the parameters of the modules frozen.

A related approach used in classifier networks is creating optimal independent two-class classifiers as building blocks for a larger network. Assume that we are trying to construct a network that can classify speech data as corresponding to one of 10 different phonemes. We can start training hidden units that learn to separate one phoneme from another. Since the number of different pairs of phonemes is 45, we have to train 45 hidden units. Training proceeds rapidly, then the 45 units are put together in the hidden layer. The output units (which receive inputs from the hidden units and from the original data) can be trained one by one. As in cascade correlation, we do not need to train more than a single layer of weights at each step. Moreover, if a new category is introduced, for example a new phoneme, we just have to train new hidden units for this new class and those already existing, add the new units to the hidden layer, and retrain the output layer.

Note that the classifiers constructed using this approach still satisfy the conditions necessary to potentially transform them into probability estimators. The proof of Proposition 13 is completely general: no requirements are put on the form of the classifier network other than enough plasticity.

A more general framework is the one proposed by Jacobs and Jordan with their *mixture of experts* approach [218]. Given a training set consisting of p pairs $(\mathbf{x}_1, \mathbf{y}_1) \ldots, (\mathbf{x}_p, \mathbf{y}_p)$, the complete network is built using several expert subnetworks. Each one of them receives the input $\mathbf{x}$ and generates the output $\mathbf{y}$ with probability

$$P(\mathbf{y}|\mathbf{x}, \theta_i)$$

where θ_i is the parameter vector of the i-th expert. The output of each expert subnetwork is weighted by a gating subnetwork which inspects the input $\mathbf{x}$ and produces for m expert subnetworks the set of gating values $g_1, g_2, \ldots, g_m$. They represent the probability that each expert network is producing the correct result. The final probability of producing $\mathbf{y}$ is given by the product of each expert's evaluation and its corresponding gating weight g_i. The parameters of the combined model can be found by gradient descent or using an Expectation Maximization (EM) algorithm proposed by Jordan and Jacobs [227]. The mixtures of experts can be organized in different levels, like an optimal decision tree. The resulting architecture is called a *Hierarchical Mixture of Experts* (HME).

16.2 Hybrid networks

In this section we consider some of the more promising examples of combined networks, especially those which couple a self-organizing with a feed-forward layer.

16.2.1 The ART architectures

Grossberg has proposed several different hybrid models, which should come closer to the biological paradigm than pure feed-forward networks or standard associative memories. His best-known contribution is the family of ART architectures (*Adaptive Resonance Theory*). The ART networks classify a stochastic sequence of vectors in clusters. The dynamics of the network consists of a series of automatic steps that resemble learning in humans. Specially phenomena like one-shot learning can be recreated with this model.

Figure 16.2 shows the task to be solved by the network. The weight vectors $\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_m$ represent categories in input space. All vectors located inside the cone around each weight vector are considered members of a specific cluster. The weight vectors are associated with computing units in a network.

Each unit fires a 1 only for vectors located inside its associated cone of radius r. The value r is inversely proportional to the "attention" parameter of the unit. If r is small, the classification of input space is fine grained. A large r produces a classification with low granularity, that is, with large clusters. The network is trained to find the weight vectors appropriate for a given data set.

Once the weight vectors have been found, the network computes whether new data can be classified in the existing clusters. If this is not the case (if, for example, a new vector is far away from any of the existing weight vectors) a new cluster is created, with a new associated weight vector. Figure 16.2 shows the weight vectors $\mathbf{w}_1$ and $\mathbf{w}_2$ with their associated cones of radius r. If the new vector $\mathbf{w}_3$ is presented to the network, it is not recognized and a new cluster is created, with this vector in the middle. The network preserves its *plasticity*, because it can always react to unknown inputs, and its *stability*, because already existing clusters are not washed out by new information. For the scheme to work, enough potential weight vectors, i.e., computing units, must be provided.

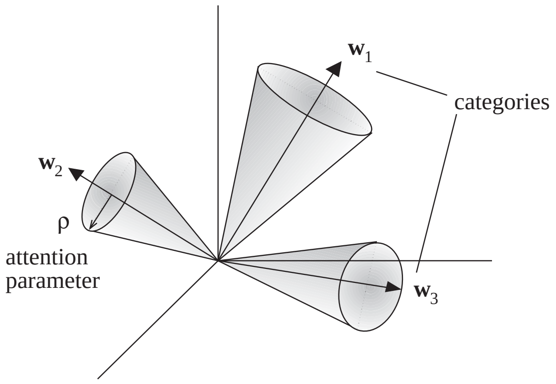


Fig. 16.2. Vector clusters and attention parameters

The ART-1 architecture was proposed by Grossberg to deal with binary vectors [168]. It was generalized, as ART-2, to real-valued vectors [81]. Later it was extended into the ART-3 model, whose dynamics is determined by differential equations similar to the ones involved in the description of chemical signals [82]. All three basic models have a similar block structure and the classification strategy is in principle the same. In what follows, we give only a simplified description of the ART-1 model. Some details have been omitted from the diagram to simplify the discussion.

Figure 16.3 shows the basic structure of ART-1. There are two basic layers of computing units. Layer F_2 contains elements which fire according to the "winner-takes-all" method. Only the element receiving the maximal scalar product of its weight vector and the input fires a 1. In our example, the

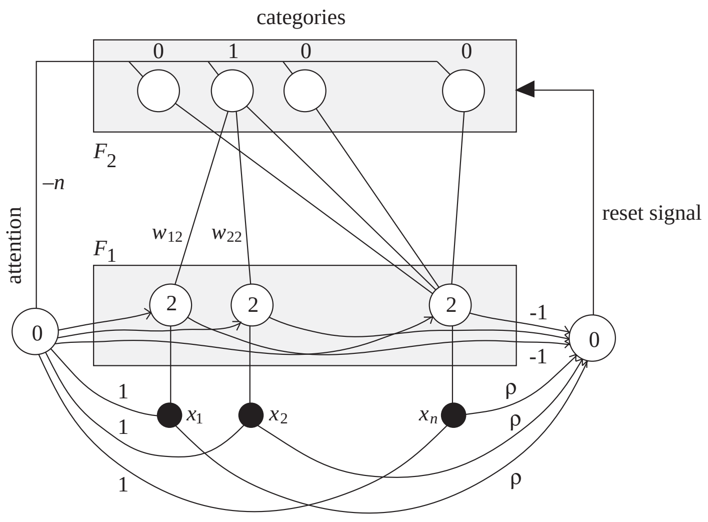


Fig. 16.3. The ART-1 architecture

second element from the left in the F_2 layer has just produced the output 1 and the other units remain silent. All weight vectors of the units in the F_2 layer are equal at the beginning (all components set to 1) and differentiate gradually as learning progresses.

Layer F_1 receives binary input vectors from the input sites. The units here have all threshold 2. This means that the attention unit (left side of the diagram) can only be turned off by layer F_2. As soon as an input vector arrives it is passed to layer F_1 and from there to layer F_2. When a unit in layer F_2 has fired, the negative weight turns off the attention unit. Additionally, the winning unit in layer F_2 sends back a 1 through the connections between layer F_2 and F_1. Now each unit in layer F_1 becomes as input the corresponding component of the input vector $\mathbf{x}$ and of the weight vector $\mathbf{w}$. The i-th F_1 unit compares x_i with w_i and outputs the product $x_i w_i$. The reset unit receives this information and also the components of $\mathbf{x}$, weighted by ρ, the attention parameter, so that it own computation is

$$\rho \sum_{i=1}^{n} x_i - \mathbf{x} \cdot \mathbf{w} \geq 0.$$

This corresponds to the test

$$\frac{\mathbf{x} \cdot \mathbf{w}}{\sum_{i=1}^{n} x_i} \leq \rho.$$

The reset unit fires only if the input lies outside the attention cone of the winning unit. A reset signal is sent to layer F_2, but only the winning unit is inhibited. This in turn activates again the attention unit and a new round of computation begins.

As we can see there is *resonance* in the network only if the input lies close enough to a weight vector $\mathbf{w}$. The weight vectors in layer F_2 are initialized with all components equal to 1 and ρ is selected to satisfy $0 < \rho < 1$. This guarantees that eventually an unused vector will be recruited to represent a new cluster. The selected weight vector $\mathbf{w}$ is updated by pulling it in the direction of $\mathbf{x}$. This is done in ART-1 by turning off all components in $\mathbf{w}$ which are zeroes in $\mathbf{x}$. In ART-2 the update step corresponds to a rotation of $\mathbf{w}$ in the direction of $\mathbf{x}$.

The purpose of the reset signal is to inhibit all units that do not resonate with the input. A unit in layer F_2, which is still unused, can be selected for the new cluster containing $\mathbf{x}$. In this way, a single presentation of an example sufficiently different from previous data, can lead to a new cluster. Modifying the value of the attention parameter ρ we can control how many clusters are used and how wide they are.

In this description we have assumed that the layer F_2 can effectively compute the angular distance between $\mathbf{x}$ and $\mathbf{w}$. This requires a normalization of $\mathbf{w}$, which can be made in substructures of layers F_1 or F_2, like for example a so-called *Grossberg layer* [168, 169].

The dynamics of the ART-2 and ART-3 models is governed by differential equations. Computer simulations consume too much time. Consequently, implementations using analog hardware or a combination of optical and electronic elements are more suited to this kind of model [462].

16.2.2 Maximum entropy

The strategy used by the ART models is only one of the many approaches that have been proposed in the literature for the clustering problem. It has the drawback that it tries to build clusters of the same size, independently of the distribution of the data. A better solution would be to allow the clusters to have varying radius (attention parameters in Grossberg terminology). A popular approach in this direction is the maximum entropy method.

The entropy H of a data set of N points assigned to k different clusters $c_1, c_2, \ldots, c_k$ is given by

$$H = -\sum_{i=1}^{k} p(c_i) \log(p(c_i)),$$

where $p(c_i)$ denotes the probability of hitting the i-th cluster, when an element of the data set is picked at random. If all data is assigned to one cluster, then $H = 0$. Since the probabilities add up to 1, the clustering that maximizes the entropy is the one for which all cluster probabilities are identical. This means that the clusters will tend to cover the same number of points.

A problem arises whenever the number of elements of each class is different in the data set. Consider the case of unlabeled speech data: some phonemes are more frequent than others and if a maximum entropy method is used, the boundaries between clusters will deviate from the natural solution and classify some data erroneously. Figure 16.4 illustrates this problem with a simple example of three clusters. The natural solution seems obvious; the maximum entropy partition is the one shown.

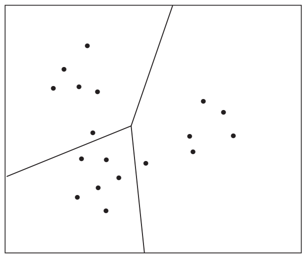


Fig. 16.4. Maximum entropy clustering

This problem can be solved using a *bootstrapped* iterative algorithm, consisting of the following steps:

Algorithm 16.2.1 *Bootstrapped clustering*

cluster: Compute a maximum entropy clustering with the training data. Label the *original data* according to this clustering.

select: Build a new training set by selecting from each class the same number of points (random selection with replacement). Go to the previous step.

The training set for the first iteration is the original data set. Using this algorithm, the boundaries between clusters are adjusted irrespective of the number of members of each actual cluster and a more natural solution is found.

16.2.3 Counterpropagation networks

Our second example of a hybrid model is that of counterpropagation networks, first proposed by Hecht-Nielsen [183]. In a certain sense, they are an extension of Kohonen's approach. The original counterpropagation networks were designed to approximate a continuous mapping f and its inverse f^{-1} using two symmetric sections. We describe the essential elements needed to learn an approximation of the mapping f.

Figure 16.5 shows the architecture of a counterpropagation network. The input consists of an n-dimensional vector which is fed to a a hidden layer consisting of h cluster vectors. The output layer consists of a single linear associator (when learning a mapping $f : \mathbb{R}^n \rightarrow \mathbb{R}$). The weights between hidden layer and output unit are adjusted using supervised learning.

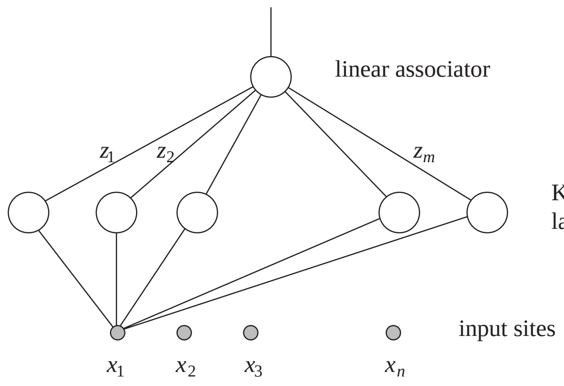


Fig. 16.5. Simplified counterpropagation network

The network learns in two different phases. Firstly, the hidden layer is trained using stochastically selected vectors from input space. The hidden layer produces a clustering of input space that corresponds to an n-dimensional Voronoi tiling (as shown in Figure 16.6). After this step, each element of the hidden layer has specialized to react to a certain region of input space. Note that the training strategy can be any of the known vector quantization methods and that the hidden layer can be arranged in a grid (Kohonen layer) or can consist of isolated elements. The output of the hidden layer can be controlled in such a way that only the element with the highest activation fires. The hidden layer is, in fact, a classification network.

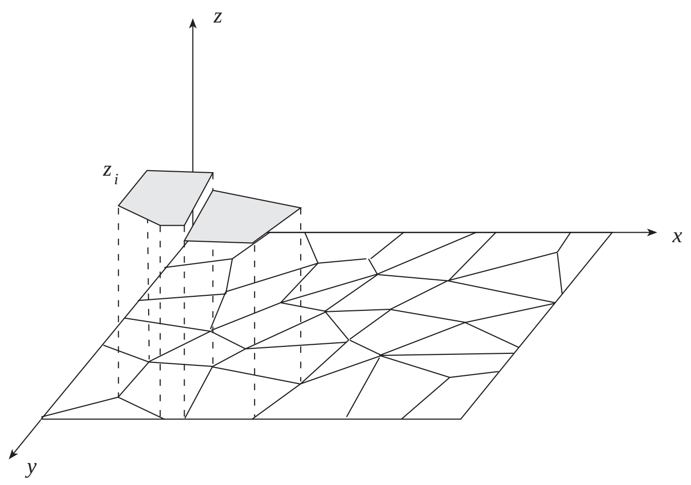


Fig. 16.6. Function approximation with a counterpropagation network

Function approximation can be implemented by assigning a value to each weight $z_1, z_2, \ldots, z_m$ in the network of Figure 16.5. This corresponds to fixing the value of the approximation for each cluster region, as shown in Figure 16.6. The polygon with height z_i in the diagram corresponds to the function approximation selected for this region. The second training step is the supervised adjustment of the z_i values. If hidden unit i fires when input vector $\mathbf{x}$ is presented, the quadratic error of the approximation is

$$E = \frac{1}{2}(z_i - f(\mathbf{x}))^2$$

Straightforward gradient descent on this cost function provides the necessary weight update

$$\Delta z_i = -\frac{dE}{dz_i} = \gamma(f(\mathbf{x}) - z_i),$$

where γ is a learning constant. After several iterations of supervised learning we expect to find a good approximation of the function f. Training of the hidden and the output layer can be intermixed, or can be made in sequence. The counterpropagation network can be trivially extended to the case of several output units.

16.2.4 Spline networks

The function approximation provided by a network with a clustering layer can be improved by computing a local linear approximation to the objective function. Figure 16.7 shows a network which has been extended with a hidden layer of linear associators. Each cluster unit is used to activate or inhibit one of the linear associators, which are connected to all input sites.

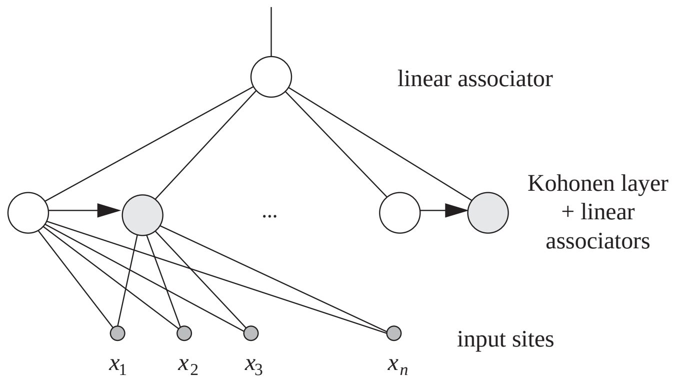


Fig. 16.7. Extended counterpropagation network

The clustering elements in the hidden layer are trained exactly as before. Figure 16.8 shows the final Voronoi tiling of input space. Now, the linear associators in the hidden layer are trained using the backpropagation algorithm. For each selected input, exclusively one of the clustering units activates one associator. Only the corresponding weights are trained. The net effect is that the network constructs a set of linear approximations to the objective function that are locally valid.

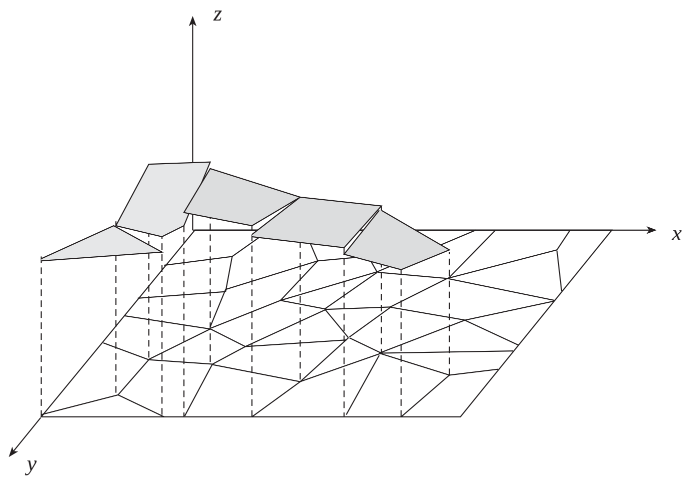


Fig. 16.8. Function approximation with linear associators

Figure 16.8 shows that the constructed approximation consists now of differently oriented polygons. The new surface has a smaller quadratic error than the approximation with horizontal tiles. The functional approximation can be made more accurate just by adding new units to the hidden layer, refining the Voronoi tiling in this way. The network can be trivially extended to the case of more than one output unit.

Ritter has recently proposed to use splines as construction elements for the function approximation. This is a generalization of the above approach, which seems to require many fewer hidden units in the case of well-behaved functions. Such *spline networks* have been applied to different pattern recognition problems and in robotics [439].

A problem that has to be addressed when combining a self-organized hidden layer with a conventional output layer, is the interaction between the two types of structures. Figure 16.9 shows an example in which the function to be learned is much more variable in one region than another. If input space is sampled homogeneously, the mean error at the center of the domain will be much larger than at the periphery, where the function is smoother.

The general solution is sampling input space according to the variability of the function, that is, according to its gradient. In this case the grid defined by, for example, a two-dimensional Kohonen network is denser at the center than

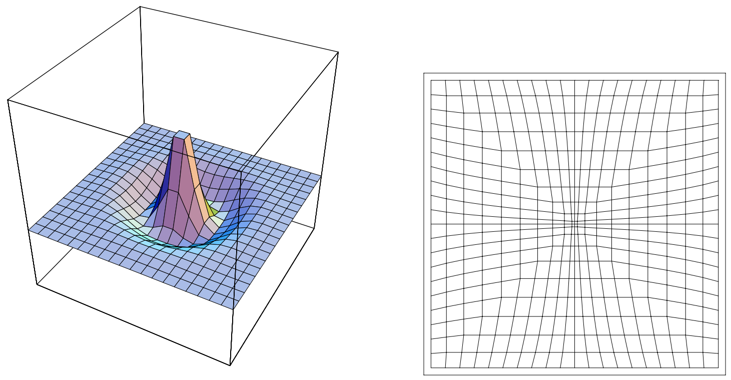


Fig. 16.9. A function and its associated differential tiling of input space

at the periphery (Figure 16.9), right). However, since only one set of training pairs is given, heuristic methods have to be applied to solve this difficulty (see Exercise 4).

16.2.5 Radial basis functions

A variant of hybrid networks with a Kohonen or clustering layer consists in using Gaussians as activation function of the units. The hidden layer is trained in the usual manner, but the input is processed differently at each hidden unit. All of them produce an output value which is combined by the linear associator at the output. It is desirable that the unit whose weight vector lies closer to the input vector fires more strongly than the other hidden units.

Moody and Darken [318] propose the architecture shown in Figure 16.10. Each hidden unit j computes as output

$$g_j(x) = \frac{\exp\left((\mathbf{x}-\mathbf{w}_j)^2/2\sigma_j^2\right)}{\sum_k \exp\left((\mathbf{x}-\mathbf{w}_k)^2/2\sigma_k^2\right)}$$

where $\mathbf{x}$ is the input vector and $\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_m$ are the weight vectors of the hidden units. The constants σ_i are selected in an ad hoc way and can be set to the distance between the weight vector $\mathbf{w}_i$ and its nearest neighbor.

Note that the output of each hidden unit is normalized by dividing it with the sum of all other hidden outputs. This explains the horizontal connections between units shown in Figure 16.10. The output of the hidden layer consists therefore of normalized numbers.

The weights $z_1, z_2, \ldots, z_m$ are determined using backpropagation. The quadratic error is given by

$$E = \frac{1}{2}(\sum_i^n g_i(\mathbf{x})z_i - f(\mathbf{x}))^2.$$

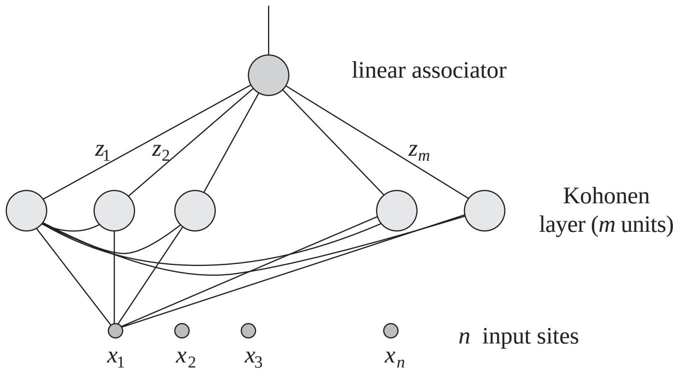


Fig. 16.10. Hybrid network with RBFs

The necessary weight updates are therefore given by

$$\Delta z_i = -\frac{dE}{dz_i} = \gamma g_i(\mathbf{x})(f(\mathbf{x}) - \sum_i^n g_i(\mathbf{x})z_i).$$

The mixture of Gaussians provides a continuous approximation of the objective function and makes unnecessary the computation of the maximum excitation in the hidden layer. Figure 16.11 shows an example of a function approximation with four Gaussians. The error can be minimized using more hidden units.

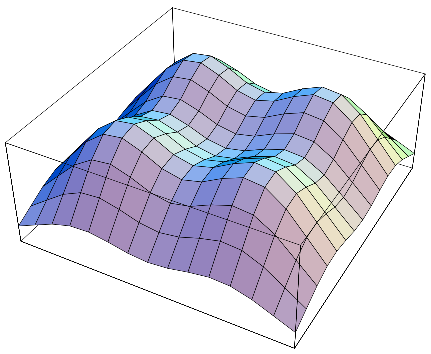


Fig. 16.11. Function approximation with RBFs

The main difference between networks made of radial basis functions and networks of sigmoidal units is that the former use locally concentrated functions as building blocks whereas the latter use smooth steps. If the function to be approximated is in fact a Gaussian, we need to arrange several sigmoidal steps in order to delimit the region of interest. But if the function to be approximated is a smooth step, we need many Gaussians to cover the region of interest. Which kind of activation function is more appropriate therefore depends on the concrete problem at hand.

16.3 Historical and bibliographical remarks

To the extent that neural network research has been approaching maturity, more sophisticated methods have been introduced to deal with complex learning tasks. Modular neural networks were used from the beginning, for example in the classic experiments of Rosenblatt. However, the probabilistic and algorithmic machinery needed to handle more structured networks has been produced only in the last few years.

The original motivation behind the family of ART models [168] was investigating phenomena like short-term and long-term memory. ART networks have grown in complexity and combine several layers or submodules with different functions. They can be considered among the first examples of the modular approach to neural network design.

Feldman et al. have called the search for algorithmic systems provided with the "speed, robustness and adaptability typically found in biology" the "grand challenge" of neural network research [137]. The path to its solution lies in the development of *structured connectionist architectures*. Biology provides here many sucessful examples of what can be achieved when modules are first found and then combined in more complex systems.

Ultimately, the kind of combined systems we would like to build are those in which symbolic rules are implemented and supported by a connectionist model [135, 136]. There have been several attempts to build connectionist experts systems [151] and also connectionist reasoning systems [400]. A modern example of a massively parallel knowledge based reasoning system is SHRUTI, a system proposed by Shastri, and which has been implemented in parallel hardware [289].

Exercises

1. Show how to eliminate the absolute value operation from equation (16.1) in the cascade correlation algorithm, so that backpropagation can be applied as usual.
2. Compare the cascade correlation algorithm with standard backpropagation. Use some small problems as benchmarks (parity, decoder-encoder, etc.).
3. Construct a counterpropagation network that maps the surface of a sphere onto the surface of a torus (represented by a rectangle). Select the most

appropriate coordinates. How must you train the network so that angles are preserved? This is the Mercator projection used for navigation.

4. Propose a method to increase sampling in those regions of input space in which the training set is more variable, in order to make a counterpropagation network more accurate.

17

Genetic Algorithms

17.1 Coding and operators

Learning in neural networks is an optimization process by which the error function of a network is minimized. Any suitable numerical method can be used for the optimization. Therefore it is worth having a closer look at the efficiency and reliability of different strategies. In the last few years genetic algorithms have attracted considerable attention because they represent a new method of stochastic optimization with some interesting properties [163, 305]. With this class of algorithms an evolution process is simulated in the computer, in the course of which the parameters that produce a minimum or maximum of a function are determined. In this chapter we take a closer look at this technique and explore its applicability to the field of neural networks.

17.1.1 Optimization problems

Genetic algorithms evaluate the target function to be optimized at some randomly selected points of the definition domain. Taking this information into account, a new set of points (a new population) is generated. Gradually the points in the population approach local maxima and minima of the function. Figure 17.1 shows how a population of points encloses a local maximum of the target function after some iterations. Genetic algorithms can be used when no information is available about the gradient of the function at the evaluated points. The function itself does not need to be continuous or differentiable. Genetic algorithms can still achieve good results even in cases in which the function has several local minima or maxima.

These properties of genetic algorithms have their price: unlike traditional random search, the function is not examined at a single place, constructing a possible path to the local maximum or minimum, but many different places are considered simultaneously. The function must be calculated for all elements of the population. The creation of new populations also requires additional calculations. In this way the optimum of the function is sought in several directions simultaneously and many paths to the optimum are processed in parallel. The calculations required for this feat are obviously much more extensive than for a simple random search.

However, compared to other stochastic methods genetic algorithms have the advantage that they can be parallelized with little effort. Since the calculations of the function on all points of a population are independent from each other, they can be carried out in several processors [164]. Genetic algorithms are thus inherently parallel. A clear improvement in performance can be achieved with them in comparison to other non-parallelizable optimization methods.

Fig. 17.1. A population of points encircles the global maximum after some generations.

Compared to purely local methods (e.g., gradient descent) genetic algorithms have the advantage that they do not necessarily remain trapped in a suboptimal local maximum or minimum of the target function. Since information from many different regions is used, a genetic algorithm can move away from a local maximum or minimum if the population finds better function values in other areas of the definition domain.

In this chapter we show how evolutionary methods are used in the search for minima of the error function of neural networks. Such error functions have special properties which make their optimization difficult. We will discuss the extent to which genetic algorithms can overcome these difficulties.

Even without this practical motivation the analysis of genetic algorithms is important, because in the course of evolution the networking pattern of biological neural networks has been created and improved. Through an evolutionary organization process nerve systems were continuously modified until they attained an enormous complexity. Therefore, by studying artificial neural networks and their relationship with genetic algorithms we can gain further valuable insights for understanding biological systems.

17.1.2 Methods of stochastic optimization

Let us look at the problem of the minimization of a real function f of the n variables $x_1, x_2, \ldots, x_n$. If the function is differentiable the minima can often be found by direct analytical methods. If no analytical expression for f is known or if f is not differentiable, the function can be optimized with stochastic methods. The following are some well-known techniques:

Random search

The simplest form of random optimization is stochastic search. A starting point $x = (x_1, x_2, \ldots, x_n)$ is randomly generated and $f(x_1, x_2, \ldots, x_n)$ is computed. Then a direction is sought in which the value of the function decreases. To do this a vector $\boldsymbol{\delta} = (\delta_1, \ldots, \delta_n)$ is randomly generated and f is computed at $(x_1 + \delta_1, \ldots, x_n + \delta_n)$. If the value of the function at this point is lower than at $x = (x_1, x_2, \ldots, x_n)$ then $(x_1 + \delta_1, \ldots, x_n + \delta_n)$ is taken as the new search point and the algorithm is started again. If the new function value is greater, a new direction is generated. The algorithm runs (within a predetermined maximum number of attempts) until no further decreasing direction of function values can be found.

The algorithm can be further improved by making the length of the direction vector $\boldsymbol{\delta}$ decrease in time. Thus the minimum of the function is approximated by increasingly smaller steps.

Fig. 17.2. A local minimum traps the search process

The disadvantage of simple stochastic search is that a local minimum of the function can steer the search in the wrong direction (Figure 17.2). However, this can be partially compensated by carrying out several independent searches.

Metropolis algorithm

Stochastic search can be improved using a technique proposed by Metropolis. ([304]). If a new search direction $(\delta_1, \ldots, \delta_n)$ guarantees that the function value decreases, it is used to update the search position. If the function in this direction increases, it is still used with the probability p where

$$p = \frac{1}{1 + \exp\left[\frac{1}{\alpha}\left(f(x_1 + \delta_1, \ldots, x_n + \delta_n) - f(x_1, \ldots, x_n)\right)\right]}$$

The constant α approaches zero gradually. This means that the probability p tends towards zero if the function f increases in the direction $(\delta_1, \ldots, \delta_n)$. In the final iterations of the algorithm only those directions in which the function values decrease are actually taken.

This strategy can prevent the iteration process from remaining trapped in suboptimal minima of the function f. With probability $p > 0$ an iteration can take an ascending direction and possibly overcome a local minimum.

Bit-based descent methods

If the problem can be recoded so that the function f is calculated with the help of a binary string (a sequence of binary symbols), then bit-based stochastic methods can be used. For example, let f be the one-dimensional function $x \mapsto (1 - x)^2$. The positive real value x can be coded in a computer as a binary number. The fixed-point coding of x in eight bits

$$x = b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$$

can be interpreted as follows: the first three bits b_7, b_6 and b_5 code that part of the value of x which is in front of the decimal point in the binary code. The five bits b_4 to b_0 code the portion of the value of x after the point. Thus only numbers whose absolute value is less than 8 are coded. An additional bit can be used for the sign.

With this recoding f is a real function over all binary strings with length eight. The following algorithm can be used: a randomly chosen initial string $b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$ is generated. The function f is then computed for $x = b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$. A bit of the string is selected at random and flipped. Let the new string be, for example, $x' = b_7' b_6 b_5 b_4 b_3 b_2 b_1 b_0$. The function f is computed at this new point. If the value of the function is lower than before, the new string is accepted as the current string and the algorithm is started again. The algorithm runs until no bit flip improves the value of the function [103].

Strictly speaking this algorithm is only a special instance of stochastic search. The only difference is that the directions which can be generated are now fixed from the beginning. There are only eight possibilities which correspond to the eight bits of the fixed-point representation. The precision of the approximation is also fixed from the beginning because with 8-bit fixed-point coding only a certain maximum precision can be achieved. The search space of the optimization method and the possible search direction are thus discretized.

The method can be generalized for n-dimensional functions by recoding the real values $x_1, x_2, \ldots, x_n$ in n binary strings which are then appended to form a single string, which is processed by the algorithm. The Metropolis strategy can also be used in bit-based methods.

Bit-based optimization techniques are already very close to genetic algorithms; these naive search methods work effectively in many cases and can even outdo elaborate genetic algorithms [103]. If the function to be optimized is not too complex, they reach the optimal minimum with substantially fewer iteration steps than the more intricate algorithms.

17.1.3 Genetic coding

Genetic algorithms are stochastic search methods managing a population of simultaneous search positions. A conventional genetic algorithm consists of three essential elements:

- a coding of the optimization problem
- a mutation operator
- a set of information-exchange operators

The *coding* of the optimization problem produces the required discretization of the variable values (for optimization of real functions) and makes their simple management in a population of search points possible. Normally the maximum number of search points, i.e., the *population size*, is fixed at the beginning.

The *mutation operator* determines the probability with which the data structures are modified. This can occur spontaneously (as in stochastic search) or only when the strings are combined to generate a new population of search points. In binary strings a mutation corresponds to a bit flip.

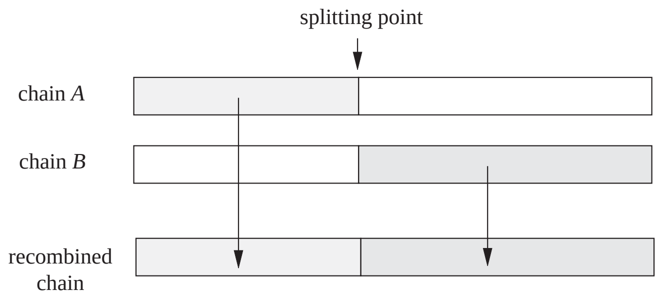


Fig. 17.3. An example of crossover

The *information exchange operators* control the recombination of the search points in order to generate a new, better population of points at each iteration step. Before recombining, the function to be optimized must be evaluated for all data structures in the population. The search points are then sorted in the order of their function value, i.e., in the order of their so-called *fitness*. In a minimization problem the points which are placed at the beginning of the list are those for which the function value is lowest. Those points for which the function to be minimized has the greatest function value are placed at the end of the list. Following this sorting operation the points are reproduced in such a way that the data structures at the beginning of the list are selected with a higher probability than the ones at the end of the list. A typical reproduction operator is *crossover*. Two strings A and B are selected as "parents" and a cut-off position for both is selected at random. The new string is formed so that the left side comes from one parent and the right side from the other. This produces an interchange of the information stored in each parent string. The whole process is reminiscent of genetic exchange in living organisms. A favorable interchange can produce a string closer to the minimum of the target function than each parent string by itself. We expect the collective fitness of the population to increase in time. The algorithm can be stopped when the fitness of the best string in the population no longer changes.

For a more precise illustration of genetic algorithms we will now consider the problem of string coding.

Optimization problems whose variables can be coded in a string are suitable for genetic algorithms. To this end an alphabet for the coding of the information must be agreed upon. Consider, for example, the following problem: a keyboard is to be optimized by distributing the 26 letters A to Z over 26 positions. The learning time of various selected candidates is to be minimized. This problem could only be solved by a multitude of experiments. A suitable coding would be a string with 26 symbols, each of which can be one of the 26 letters (without repeats). So the alphabet of the coding consists of 26 symbols, and the search space of the problem contains 26! different combinations. For Asian languages the search space is even greater and the problem correspondingly more difficult [162].

A binary coding of the optimization problem is ideal because in this way the mutation and information exchange operators are simple to implement. With neural networks, in which the parameters to be optimized are usually real numbers, the definition of an adequate coding is an important problem. There are two alternatives: floating-point or fixed-point coding. Both possibilities can be used, whereby fixed-point coding allows more gradual mutations than floating-point coding [221]. With the latter the change of a single bit in the exponent of the number can cause a dramatic jump. Fixed-point coding is usually sufficient for dealing with the parameters of neural networks (see Sect. 8.2.3).

17.1.4 Information exchange with genetic operators

Genetic operators determine the way new strings are generated out of existing ones. The mutation operator is the simplest to describe. A new string can be generated by copying an old string position by position. However, during copying each symbol in the string can be modified with a certain probability, the mutation rate. The new string is then not a perfect copy and can be used as a new starting point for a search operation in the definition domain of the function to be optimized.

The crossover operator was described in the last section. With this operator portions of the information contained in two strings can be combined. However an important question for crossover is at which exact places we are allowed to partition the strings.

Both types of operator can be visualized in the case of function optimization. Assume that the function $x \mapsto x^2$ is to be optimized in the interval $[0,1]$. The values of the variable x can be encoded with the binary fixed-point coding $x = 0.b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0$. Thus there are 1024 different values of x and only one of them optimizes the proposed quadratic function. The code used discretizes the definition space of the function. The minimum distance between consecutive values of x is 2^{-10}.

A mutation of the i-th bit of the string $0.b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0$ produces a change of x of $\delta = 2^{-i}$. Thus a new point $x + \delta$ is generated. In this case the mutation corresponds to a stochastic search operation with variable step length.

When the number $x = 0.b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0$ is recombined with the number $y = 0.a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$, a cut-off position i is selected at random. The new string, which belongs to the number z, is then:

$$z = 0.b_9b_8 \cdots b_i a_{i-1} \cdots a_0.$$

The new point z can also be written as $z = x + \delta$, where δ is dependent on the bit sequences of x and y. Crossover can therefore be interpreted as a further variation of stochastic search. But note that in this extremely simplified example any gradient descent method is much more efficient than a genetic algorithm.

17.2 Properties of genetic algorithms

Genetic algorithms have made a real impact on all those problems in which there is not enough information to build a differentiable function or where the problem has such a complex structure that the interplay of different parameters in the final cost function cannot be expressed analytically.

17.2.1 Convergence analysis

The advantages of genetic algorithms first become apparent when a population of strings is observed. Let f be the function $x \mapsto x^2$ which is to be maximized, as before, in the interval $[0, 1]$. A population of N numbers in the interval $[0, 1]$ is generated in 10-bit fixed-point coding. The function f is evaluated for each of the numbers $x_1, x_2, \ldots, x_N$, and the strings are then listed in descending order of their function values. Two strings from this list are always selected to generate a new member of a new population, whereby the probability of selection decreases monotonically in accordance with the ascending position in the sorted list.

The computed list contains N strings which, for a sufficiently large N, should look like this:

$$\begin{array}{l} 1\ 0.1{*}{*}{*}{*}{*}{*}{*}{*}{*}{*} \\ 1\ 0.1{*}{*}{*}{*}{*}{*}{*}{*}{*}{*} \\ \vdots\ \ \vdots \\ 1\ 0.0{*}{*}{*}{*}{*}{*}{*}{*}{*}{*} \end{array}$$

The first positions in the list are occupied by strings in which the first bit after the point is a 1 (i.e., the corresponding numbers lie in the interval $[0.5, 1]$). The last positions are occupied by strings in which the first bit after the decimal point is a 0. The asterisk stands for any bit value from 0 to 1, and the zero in front of the point does not need to be coded in the strings. The upper strings are more likely to be selected for a recombination, so that the offspring is more likely to contain a 1 in the first bit than a 0. The new population is evaluated and a new fitness list is drawn up. On the other hand, strings with a 0 in the first bit are placed at the end of the list and are less likely to be selected than the strings which begin with a 1. After several generations no more strings with a 0 in the first bit after the decimal point are contained in the population.

The same process is repeated for the strings with a zero in the second bit. They are also pushed towards extinction. Gradually the whole population converges to the optimal string 0.1111111111 (when no mutation is present).

With this quadratic function the search operation is carried out within a very well-ordered framework. New points are defined at each crossover, but steps in the direction $x = 1$ are more likely than steps in the opposite direction. The step length is also reduced in each reproduction step in which the 0 bits are eliminated from a position. When, for example, the whole population only consists of strings with ones in the first nine positions, then the maximum step length can only be 2^{-10}.

The whole process strongly resembles simulated annealing. There, stochastic jumps are also generated, whereby transitions in the maximization direction are more probable. In time the temperature constant decreases to zero, so that the jumps become increasingly smaller until the system *freezes* at a local maximum.

John Holland [195] suggested the notion of *schemata* for the convergence analysis of genetic algorithms. Schemata are bit patterns which function as representatives of a set of binary strings. We already used such bit patterns in the example above: the bit patterns can contain each of the three symbols 0, 1 or $*$. The schema $**00**$, for example, is a representative of all strings of length 6 with two zeros in the central positions, such as: 100000, 110011, 010010, etc.

During the course of a genetic algorithm the best bit patterns are gradually selected, i.e., those patterns which minimize/maximize the value of the function to be optimized. Normal genetic algorithms consist of a finite repetition of the three steps:

1. selection of the parent strings,
2. recombination,
3. mutation.

This raises the question: how likely is it that the better bit patterns survive from one generation of a genetic algorithm to another? This depends on the probability with which they are selected for the generation of new child strings and with which they survive the recombination and mutation steps. We now want to calculate this probability.

In the algorithm to be analyzed, the population consists of a set of N binary strings of length ℓ at time t. A string of length ℓ which contains one of the three symbols 0, 1, or $\square$ in each position is a bit pattern or schema. The symbol $\square$ represents a 0 or a 1. The number of strings in the population in generation t which contain the bit pattern H is called $o(H,t)$. The diameter of a bit pattern is defined as the length of the pattern's shortest substring that still contains all fixed bits in the pattern. For example, the bit pattern $**1*1**$ has diameter three because the shortest fragment that contains both constant bits is the substring $1*1$ and its length is three. The diameter of a bit pattern H is called $d(H)$, with $d(H) \geq 1$. It is important to understand that a schema is of the same length as the strings that compose the population.

Let us assume that f has to be maximized. The function f is defined over all binary strings of length ℓ and is called the fitness of the strings. Two parent strings from the current population are always selected for the creation of a new string. The probability that a parent string H_j will be selected from N strings $H_1, H_2, \ldots, H_N$ is

$$p(H_j) = \frac{f(H_j)}{\sum_{i=1}^{N} f(H_i)}.$$

This means that strings with greater fitness are more likely to be selected than strings with lesser fitness. Let f_μ be the average fitness of all strings in the population, i.e.,

$$f_\mu = \frac{1}{N}\sum_{i=1}^{N} f(H_i).$$

The probability $p(H_j)$ can be rewritten as

$$p(H_j) = \frac{f(H_j)}{Nf_\mu}.$$

The probability that a schema H will be passed on to a child string can be calculated in the following three steps:

i) Selection

Selection with replacement is used, i.e., the whole population is the basis for each individual parent selection. It can occur that the same string is selected twice. The probability P that a string is selected which contains the bit pattern H is:

$$P = \frac{f(H_1)}{Nf_\mu} + \frac{f(H_2)}{Nf_\mu} + \cdots + \frac{f(H_k)}{Nf_\mu},$$

where $H_1, H2, \ldots, Hk$ represent all strings of the generation which contain the bit pattern H. If there are no such strings, then P is zero.

The fitness $f(H)$ of the bit pattern H in the generation t is defined as

$$f(H) = \frac{f(H_1) + f(H_2) + \cdots + f(H_k)}{o(H,t)}.$$

Thus P can be rewritten as

$$P = \frac{o(H,t)f(H)}{Nf_\mu}.$$

The probability P_A that two strings which contain pattern H are selected as parent strings is given by

$$P_A = \left(\frac{o(H,t)f(H)}{Nf_\mu}\right)^2.$$

The probability P_B that from two selected strings only one contains the pattern H is:

$$P_B = 2\frac{o(H,t)f(H)}{Nf_\mu}\left(1 - \frac{o(H,t)f(H)}{Nf_\mu}\right).$$

ii) Recombination

For the recombination of two strings a cut-off point is selected between the positions 1 and $\ell - 1$ and then a crossover is carried out. The probability W that a schema H is transmitted to the new string depends on two cases. If both parent strings contain H, then they pass on this substring to the new string. If only one of the strings contains H, then the schema is inherited at most half of the time. The substring H can also be destroyed with probability $(d(H)-1)/(\ell-1)$ during crossover. This means that

$$W \geq \left(\frac{o(H,t)f(H)}{Nf_\mu}\right)^2 + \frac{2}{2}\frac{o(H,t)f(H)}{Nf_\mu}\left(1-\frac{o(H,t)f(H)}{Nf_\mu}\right)\left(1-\frac{d(H)-1}{\ell-1}\right).$$

The probability W is greater than or equal to the term on the right in the above inequality, because in some favorable cases the bit string is not destroyed by crossover (one parent string contains H and the other parent *part* of H). To simplify the discussion we will not examine all these possibilities. The inequality for W can be further simplified to

$$W \geq \frac{o(H,t)f(H)}{Nf_\mu}\left(1-\frac{d(H)-1}{\ell-1}\left(1-\frac{o(H,t)f(H)}{Nf_\mu}\right)\right).$$

iii) Mutation

When two strings are recombined, the information contained in them is copied bit by bit to the child string. A mutation can produce a bit flip with the probability p. This means that a schema H with $b(H)$ fixed bits will be preserved after copying with probability $(1-p)^{b(H)}$. If a mutation occurs the probability W of the schema H being passed on to a child string changes according to W', where

$$W' \geq \frac{o(H,t)f(H)}{Nf_\mu}\left(1-\frac{d(H)-1}{\ell-1}\left(1-\frac{o(H,t)f(H)}{Nf_\mu}\right)\right)(1-p)^{b(H)}.$$

If in each generation N new strings are produced, the expected value of the number of strings which contain H in the generation $t+1$ is NW', that is

$$\langle o(H,t+1)\rangle \geq \frac{o(H,t)f(H)}{f_\mu}\left(1-\frac{d(H)-1}{\ell-1}\left(1-\frac{0(H,t)f(H)}{Nf_\mu}\right)\right)(1-p)^{b(H)}. \tag{17.1}$$

Equation (17.1) or slight variants thereof are known in the literature by the name "schema theorem" [195, 163]. This result is interpreted as stating that in the long run the best bit patterns will diffuse to the whole population.

17.2.2 Deceptive problems

The schema theorem has to be taken with a grain of salt. There are some functions in which finding the optimal bit patterns can be extremely difficult for a genetic algorithm. This happens in many cases when the optimal bits in the selected coding exhibit some kind of correlation. Consider the following function of n variables,

$$(x_1, x_2, \ldots, x_n) \mapsto \frac{x_1^2+\cdots+x_n^2}{x_1^2+\varepsilon} + \cdots + \frac{x_1^2+\cdots+x_n^2}{x_n^2+\varepsilon},$$

where ε is a small positive constant. The minimum of this function is located at the origin and any gradient descent method would find it immediately. However if the n parameters are coded in a single string using 10 bits, any time one of these parameters approaches the value zero, the value of the function increases. It is not possible to approach the origin following the direction of the axes. This means that only *correlated* mutations of the n parameters are favorable, so that the origin is reached through the diagonal valleys of the fitness function. The probability of such coordinated mutations is very small when the number of parameters n and the number of bits used to code each parameter increases. Figure 17.4 shows the shape of the two-dimensional version of this problematic function.

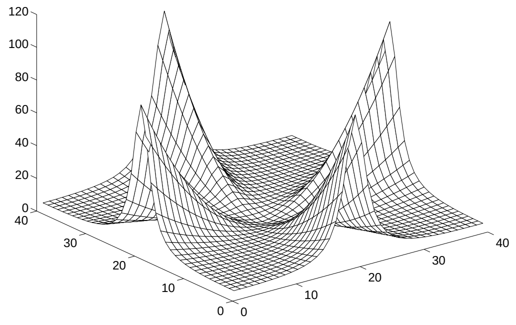


Fig. 17.4. A deceptive function

Functions which "hide" the optimum from genetic algorithms have been called *deceptive functions* by Goldberg and other authors. They mislead the GA into pursuing false leads to the optimum, which in many cases is only reached by pure luck.

There has been much research over the last few years on classifying the kinds of function which should be easy for a genetic algorithm to optimize and those which should be hard to deal with. Holland and Mitchell defined so-called "royal road" functions [314], which are defined in such a way that several parameters $x_1, x_2, \ldots, x_n$ are coded contiguously and the fitness function f is just a sum of n functions of each parameter, that is,

$$f(x_1, x_2, \ldots, x_n) = f_1(x_1) + f_2(x_2) + \cdots + f_n(x_n).$$

When these functions are optimized, genetic algorithms rapidly find the necessary values for each parameter. Mutation and crossover are both beneficial. However, mere addition of a further function f' of the variables x_1 and x_2 can slow down convergence [141]. This happens because the additional term tends to select contiguous combinations of x_1 and x_2. If the contribution from

$f'(x_1, x_2)$ to the fitness function is more important than the contribution from $f_3(x_3)$, some garbage bits at the coding positions for x_3 can become attached to the strings with the optimum values for x_1 and x_2. They then get a "free ride" and become selected more often. That is why they are called *hitch-hiking* bits [142].

Some authors have argued in favor of the *building block hypothesis* to explain why GAs do well in some circumstances. According to this hypothesis a GA finds building blocks which are then combined through the generations in order to reach the optimal solution. But the phenomena we just pointed out, and the correlations between the optimization parameters, sometimes preclude altogether the formation of these building blocks. The hypothesis has received some strong criticism in recent years [166, 141].

17.2.3 Genetic drift

Equation (17.1) giving the expected number of strings which inherit a schema H shows the following: schemata with above average fitness and small diameter will be selected more often, so that we expect them to diffuse to the population at large. The equation also shows that when the mutation rate is too high ($p \approx 1$) schemata are destroyed. The objective of having mutations is to bring variability into the population and extend it over the whole definition domain of the fitness function. New unexplored regions can be tested. This tendency to explore can be controlled by regulating the magnitude of the mutation rate. So-called *metagenetic algorithms* encode the mutation rate or the length of stochastic changes to the points in the population in the individual bit strings associated with each point. In this way the optimal mutation rate can be sought simultaneously with the optimum of the fitness function to accelerate the convergence speed.

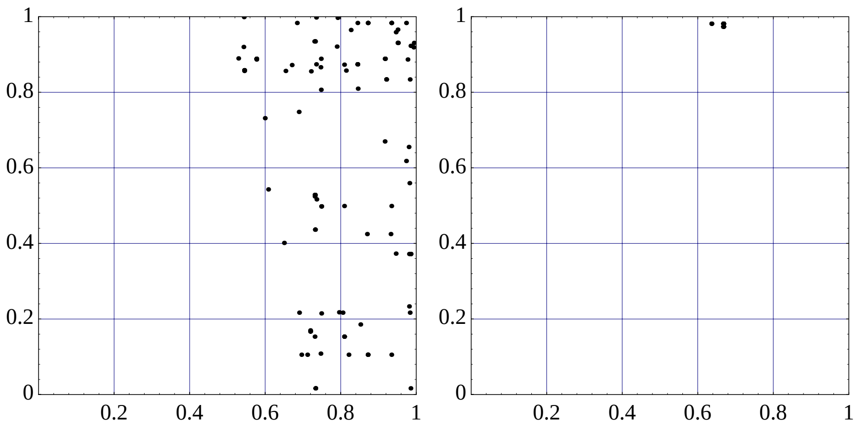


Fig. 17.5. Genetic drift in a population after 1000 and 3000 generations

Equation (17.1) also shows that when a schema H is overrepresented in a population, it can diffuse even when its fitness $f(H)$ is not different from the average fitness f_μ. In this case

$$\left(1-\frac{d(H)-1}{\ell-1}\left(1-\frac{o(H,t)f(H)}{Nf_\mu}\right)\right)=\left(1-\frac{d(H)-1}{\ell-1}\left(1-\frac{o(H,t)}{N}\right)\right).$$

Schemata with a large factor $o(H,t)/N$ will be less disturbed by crossover. The algorithm then converges to the schema H without any good reason other than the fact that the schema is already over-represented in the population. In biology this is called *genetic drift* because it represents a *random walk* in search space. Figure 17.5 shows the result of an experiment in which a population of two-dimensional points was randomly generated in the domain $[0,1]\times[0,1]$. A constant fitness function was used and no mutations were allowed, only recombinations. One could think that under these circumstances the population would expand and occupy random positions in the whole definition domain. After 100 generations the population had moved to just one side of the square and after 3000 generations it had merged to just three points. This symmetry breaking comes from an initial bias in the bit combinations present in the original population.

Losing bit patterns during the run of the GA is what a GA is all about, otherwise the population would never reach a consensus about the optimal region to be explored. This loss of information becomes problematic when the function to be optimized exhibits many flat regions. Bit patterns that would be needed for later convergence steps can be lost. Mutation tries to keep the balance between these two contradictory objectives, exploration and convergence. Finding the right mix of both factors depends on the particular problem and has to be left to the practitioner. Some alternatives are the metagenetic algorithms already mentioned or a broader set of recombination operators, such as crossover with the simultaneous inversion of one or both of the inherited string pieces. This corresponds to a massive mutation of the coded string and brings new variability into the population.

The loss of variability in the long run can be compared to the controlled lowering of the temperature constant in *simulated annealing*. In a less variable population, parent and child strings are very similar and searching is mostly done locally. Crossover by itself leads to this kind of controlled convergence to a region in search space.

17.2.4 Gradient methods versus genetic algorithms

Genetic algorithms offer some interesting properties to offset their high computational cost. We can mention at least three of them: a) GAs explore the domain of definition of the target function at many points and can thus escape from local minima or maxima; b) the function to be optimized does not need to be given in a closed analytic form – if the process being analyzed is too complex to describe in formulas, the elements of the population are used to run some experiments (numerical or in the laboratory) and the results are interpreted as their fitness; c) since evaluation of the target function is independent for each element of the population, the parallelization of the GA is quite straightforward. A population can be distributed on several processors and the selection process carried out in parallel. The kinds of recombination operator define the type of communication needed between the processors.

Straightforward parallelization and the possibility of their application in ill-defined problems makes GAs attractive. De Jong has emphasized that GAs are not function optimizers of the kind studied in numerical analysis [105]. Otherwise their range of applicability would be very narrow. As we already mentioned, in many cases a naive hill-climber is able to outperform complex genetic algorithms. The hill-climber is started n times at n different positions and the best solution is selected. A combination of a GA and hill-climbing is also straightforward: the elements in the population are selected in "Darwinian" fashion from generation to generation, but can become better by modifying their parameters in "Lamarckian" way, that is, by performing some hill-climbing steps before recombining. Davis [103] showed that many popular functions used as benchmarks for genetic algorithms can be optimized with a simple *bit climber* (stochastic bit-flipping). In six out of seven test functions the bit climber outperformed two variants of a genetic algorithm. It was three times faster than an efficient GA and twenty-three times faster than an inefficient version. Ackley [8, 9] arrived at similar results when he compared seven different optimization methods. Depending on the test function one or the other minimization strategies emerged victorious. These results only confirm what numerical analysts have known for a long time: there is no optimal optimization method, it all depends on the problem at hand. Even if this is so, the attractiveness of easy parallelization is not diminished. If no best method exists, we can at least parallelize those we have.

17.3 Neural networks and genetic algorithms

Our interest in genetic algorithms for neural networks has two sources: is it possible to use this kind of approach to find the weights in a network? And even more important: is it possible to let networks evolve so that they find an optimal topology? The question of network topology is one of those problems for which no closed-form fitness function over all possible configurations can be given. We just propose a topology and let the network run, change the topology again and look at the results. Before going into the details we have to look at some specific characteristics of neural networks which seem to preclude the use of genetic algorithms.

17.3.1 The problem of symmetries

Before genetic algorithms can be used to minimize the error function of neural networks, an appropriate code must be designed. Usually the weights of the network are floating-point numbers. Assume that the m weights $w_1, w_2, \ldots, w_m$ have been coded and arranged in a string. The target function for the GA is the error of the network for a given training set. The code for each parameter could consist of 20 bits, and in that case all network parameters could be represented by a string of $20m$ bits. These strings are then processed in the usual way.

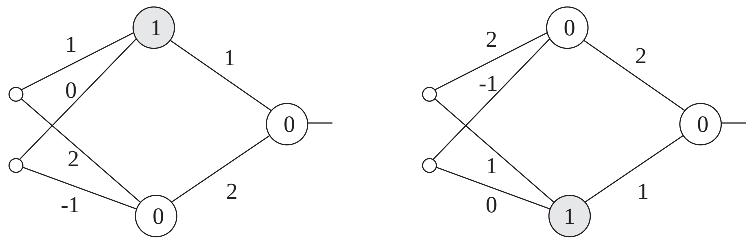


Fig. 17.6. Equivalent networks (weight permutation)

However, the target function exhibits some "deceptive" characteristics in this case. The basic problem is the high number of symmetries of the error function, which arise because the parameters of multilayered networks can be permuted without affecting the network function. The two networks shown in Figure 17.6 are equivalent, since they produce the same output for the same inputs. In both networks the functionality of the hidden units was interchanged. However, the arrangement of the coded parameters in a string looks very different. Because of the symmetries of the error function the number of local minima is high. The population of network parameters then disperses over different regions and examines different incompatible combinations. Crossover under these circumstances will almost certainly lead nowhere or converge very slowly when the population has drifted to a definite region of weight space.

Some authors' experiments have also shown that coding of the network parameters has an immediate effect on the convergence speed [317]. They have found that fixed-point coding is usually superior to floating-point coding of the parameters. Care must also be taken not to divide the strings through the middle of some parameter. It is usually better to respect the parameter boundaries so that crossover only combines the parameters but does not change the bit representation of each one.

Helping the genetic algorithm to respect good parameter combinations for a computing unit is an important problem. The simplest strategy is to find a good enumeration of the weights in a network that keeps related weights near each other in the coding string. When crossover is applied on such strings

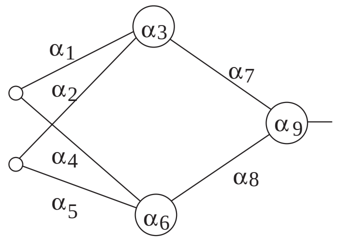


Fig. 17.7. Sequencing of the network parameters

there is a higher probability that related parameters are held together in the new string. The weights of the incoming edges into a unit constitute a logical entity and an enumeration like the one shown in Figure 17.7 can help a GA to converge on a good solution. The subindex of each weight shows which place it occupies in the coding string. This simple idea can be used with any network topology, but has obvious limits.

Modifications of the individual weights can be produced only by mutation. The mutations can be produced by copying not the value α to the child string but $\alpha+\varepsilon$, where ε is a random number of small absolute value. This produces the necessary variability of the individual parameters. The mutations produce a random walk of the network weights in weight space. The selection method of the GA separates good from bad mutations and we expect the population to stabilize at a local minimum.

17.3.2 A numerical experiment

Figure 17.8 shows a network for the encoding-decoding problem with 8 input bits. The input must be reproduced at the output, and only one of the input bits is a one, the others are zero. The hidden layer is the bottleneck for transmission of input to output. One possible solution is to encode the input in three bits at the hidden layer, but other solutions also exist.

The 48 weights of the network and the 11 bits were encoded in a string with 59 floating-point numbers. Crossover through the middle of parameters was avoided and in this case a fixed-point coding is not really necessary. Mutation was implemented as described above, not by bit flips but by adding some stochastic deviation. Figure 17.9 shows the evolution of the error curve. After 5350 generations a good parameter combination was found, capable of keeping the total error for all 8 output lines under 0.05. The figure also shows the average error for the whole population and the error for the best parameter combination in each generation.

This is an example of a small network in which the GA does indeed converge to a solution. Some authors have succeeded in training much larger networks [448]. However, as we explained before, in the case of such well-defined numerical optimization problems direct hill climbing methods are usually better.

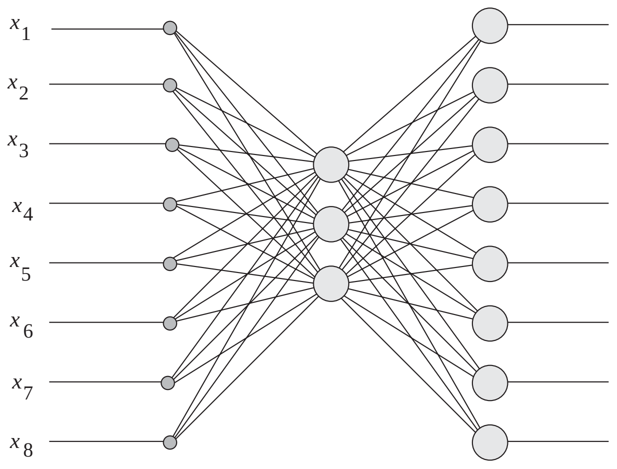


Fig. 17.8. An 8-bit encoder-decoder

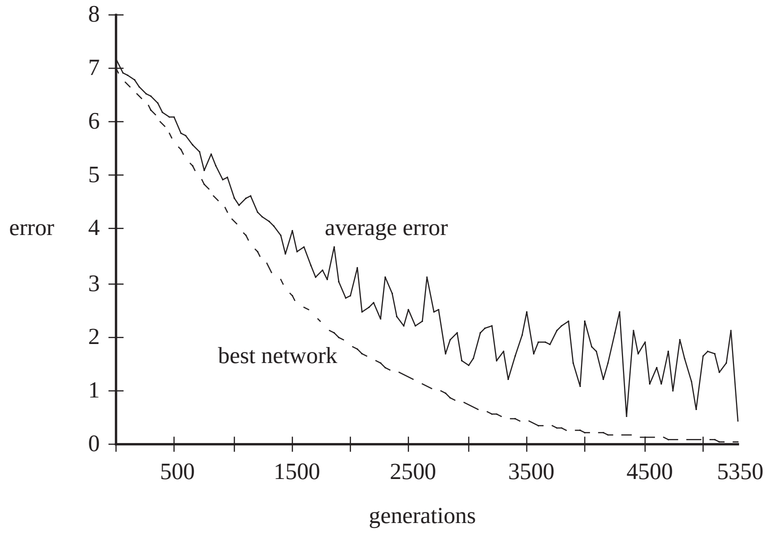


Fig. 17.9. Error function at each generation: population average and best network

Braun has shown how to optimize the network topology using genetic algorithms [69]. He introduces order into the network by defining a fitness function f made of several parts: one proportional to the approximation error, and another proportional to the *length* of the network edges. The nodes are all fixed in a two-dimensional plane and the network is organized in layers. The fitness function is minimized using a genetic algorithm. Edges with small weights are dropped stochastically. Smaller networks are favored in the course of evolution. The two-dimensional mapping is introduced to avoid destroying good solutions when reproducing the population, as happens when the network symmetries are not considered.

17.3.3 Other applications of GAs

One field in which GAs seem to be a very appropriate search method is game theory. In a typical mathematical game some rules are defined and the reward for the players after each move is given by a certain pay-off function. Mathematicians then ask what is the optimal strategy, that is, how can a player maximize his pay-off. In many cases the cumulative effect of the pay-off function cannot be expressed analytically and the only possibility is to actually play the game and compare the results using different strategies. An unknown function must be optimized and this is done with a computer simulation. Since the range of possible strategies is so large only a few variants are tested.

This is where GAs take over. A population of "players" is generated and a tournament is held. The strategy of each player must be coded in a data structure that can be recombined and mutated. Each round of the tournament is used to compute the pay-off for each strategy and the best players provide more genes for the next generation. Axelrod [36] did exactly this for the problem known as the *prisoner's dilemma*. In this game between two players, each one decides independently whether he wants to cooperate with the other player or betray him. There are four possibilities each time: a) both players cooperate; b) the first cooperates, the second betrays; c) the first betrays, the second cooperates; and c) both players betray each other. The pay-off for each of the four combinations is shown in Figure 17.10. The letters C and B stand for "cooperation" and "betrayal" respectively.

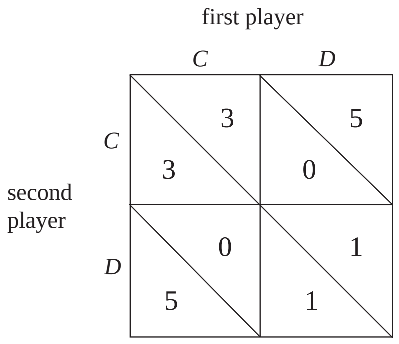


Fig. 17.10. Pay-off matrix for the prisoner's dilemma

The pay-off matrix shows that there is an incentive to commit treachery. The pay-off when one of the players betrays the other, who wants to cooperate, is 5. Moreover, the betrayed player does not get any pay-off at all. But if both players betray, the pay-off for both is just 1 point. If both players cooperate each one gets 3 points.

If the game is played only once, the optimal strategy is betraying. From the viewpoint of each player the situation is the following: if the other player cooperates, then the pay-off can be maximized by betraying. And if the other player betrays, then at least one point can be saved by committing treachery too. Since both viewpoints are symmetrical, both players betray.

However, the game becomes more complicated if it is repeated an indefinite number of times in a tournament. Two players willing to cooperate can reap larger profits than two players betraying each other. In that case the players must keep a record of the results of previous games with the same partner in order to adapt to cooperative or uncooperative adversaries. Axelrod and Hamilton [35] held such a tournament in which a population of players operated with different strategies for the iterated prisoner's dilemma. Each player could store only the results of the last three games against each opponent and the players were paired randomly at each iteration of the tournament. It was surprising that the simplest strategy submitted for the tournament collected the most points. Axelrod called it "tit for tat" (TFT) and it consists in just repeating the last move of the opponent. If the adversary cooperated the last time, cooperation ensues. If the adversary betrayed, he is now betrayed. Two players who happen to repeat some rounds of cooperation are better off than those that keep betraying. The TFT strategy is initialized by offering cooperation to a yet unknown player, but responds afterwards with vengeance for each betrayal. It can thus be exploited no more than once in a tournament.

For the first tournament, the strategies were submitted by game theorists. In a second experiment the strategies were generated in the computer by evolving them over time [36]. Since for each move there are four possible outcomes of the game and only the last three moves were stored, there are 64 possible recent histories of the game against each opponent. The strategy of each player can be coded simply as a binary vector of length 64. Each component represents one of the possible histories and the value 1 is interpreted as cooperation in the next move, whereas 0 is interpreted as betrayal. A vector of 64 ones, for example, is the coding for a strategy which always cooperates regardless of the previous game history against the opponent. After some generations of a tournament held under controlled conditions and with some handcrafted strategies present, TFT emerged again as one of the best strategies. Other strategies with a tendency to cooperate also evolved.

17.4 Historical and bibliographical remarks

At the end of the 1950s and the beginning of the 1960s several authors independently proposed the use of evolutionary methods for the solution of optimization problems. Goldberg summarized this development from the American perspective [163]. Fraser, for example, studied the optimization of polynomials, borrowing his methods from genetics. Other researchers applied GAs in the 1960s to the solution of such diverse problems as simulation, game theory, or pattern recognition. Fogel and his coauthors studied the problems of mutating populations of finite state automata and recombination, although they stopped short of using the crossover operator [139].

John Holland was the first to try to examine the dynamic of GAs and to formulate a theory of their properties [195]. His book on the subject is a classic to this day. He proposed the concept of schemata and applied statistical methods to the study of their diffusion dynamics. The University of Michigan became one of the leading centers in this field through his work.

In other countries the same ideas were taking shape. In the 1960s Rechenberg [358] and Schwefel [394] proposed their own version of evolutionary computation, which they called *evolution strategies*. Some of the problems that were solved in this early phase were, for example, hard hydrodynamic optimization tasks, such as finding the optimal shape of a rocket booster. Schwefel experimented with the evolution of the parameters of the genetic operators, maybe the first instance of metagenetic algorithms [37]. In the 1980s Rechenberg's lab in Berlin solved many other problems such as the optimal profile of wind concentrators and airplane wings.

Koza and Rice showed that it is possible to optimize the topology of neural networks [260]. Their methods make use of the techniques developed by Koza to breed Lisp programs, represented as trees of terms. Crossover exchanges whole branches of two trees in this kind of representation. Belew and coauthors [51] examined the possibility of applying evolutionary computation to connectionist learning problems.

There has been much discussion recently on the merits of genetic algorithms for static function optimization [105]. Simple minded hill-climbing algorithms seem to outperform GAs when the function to be optimized is fixed. Baluja showed in fact that for a large set of optimization problems GAs do not offer a definitive advantage when compared to other optimization heuristics, either in terms of total function evaluations or in quality of the solutions [40]. More work must be still done to determine under which situations (dynamic enviroments, for example) GAs offer a definitive advantage over other optimization methods.

The prisoner's dilemma was proposed by game-theoreticians in the 1950s and led immediately to many psychological experiments and mathematical arguments [350]. The research of Axelrod on the iterated prisoner's dilemma can be considered one of the milestones in evolutionary computation, since it opened this whole field to the social scientists and behavior biologists, who by the nature of their subject had not transcended the level of qualitative descriptions [104, 453]. The moral dilemma of the contradiction between altruism and egoism could now be modeled in a quantitative way, although Axelrod's results have not remained uncontested.

Exercises

1. How does the schema theorem (17.1) change when a crossover probability p_c is introduced? This means that with probability p_c, crossover is used to combine two strings, otherwise one of the parent strings is copied.
2. Train a neural network with a genetic algorithm. Let the number of weights increase and observe the running time.
3. Propose a coding method for a metagenetic algorithm that lets the mutation rate and the crossover probability evolve.
4. The prisoner's dilemma can be implemented on a cellular automaton. The players occupy one square of a two-dimensional grid and interact with their closest neighbors. Simulate this game in the computer and let the system evolve different strategies.

18

Hardware for Neural Networks

18.1 Taxonomy of neural hardware

This chapter concludes our analysis of neural network models with an overview of some hardware implementations proposed in recent years. In the first chapter we discussed how biological organisms process information. We are now interested in finding out how best to process information using electronic devices which in some way emulate the massive parallelism of the biological world. We show that neural networks are an attractive option for engineering applications if the *implicit* parallelism they offer can be made *explicit* with appropriate hardware. The important point in any parallel implementation of neural networks is to restrict communication to local data exchanges. The structure of some of those architectures, such as systolic arrays, resembles cellular automata.

There are two fundamentally different alternatives for the implementation of neural networks: a software simulation in conventional computers or a special hardware solution capable of dramatically decreasing execution time. A software simulation can be useful to develop and debug new algorithms, as well as to benchmark them using small networks. However, if large networks are to be used, a software simulation is not enough. The problem is the time required for the learning process, which can increase exponentially with the size of the network. Neural networks without learning, however, are rather uninteresting. If the weights of a network were fixed from the beginning and were not to change, neural networks could be implemented using any programming language in conventional computers. But the main objective of building special hardware is to provide a platform for efficient adaptive systems, capable of updating their parameters in the course of time. New hardware solutions are therefore necessary [54].

18.1.1 Performance requirements

Neural networks are being used for many applications in which they are more effective than conventional methods, or at least equally so. They have been introduced in the fields of computer vision, robot kinematics, pattern recognition, signal processing, speech recognition, data compression, statistical analysis, and function optimization. Yet the first and most relevant issue to be decided before we develop physical implementations is the size and computing power required for the networks.

The capacity of neural networks can be estimated by the number of weights used. Using this parameter, the complexity of the final implementation can be estimated more precisely than by the number of computing units. The number of weights also gives an indication of how difficult it will be to train the network. The performance of the implementation is measured in *connections per second* (cps), that is, by the number of data chunks transported through all edges of the network each second. Computing a connection requires the transfer of the data from one unit to another, multiplying the data by the edge's weight in the process. The performance of the learning algorithm is measured in *connection updates per second* (cups). Figure 18.1 shows the number of connections and connection updates per second required for some computationally intensive applications. The numbered dots represent the performance and capacity achieved by some computers when executing some of the reported neural networks applications.

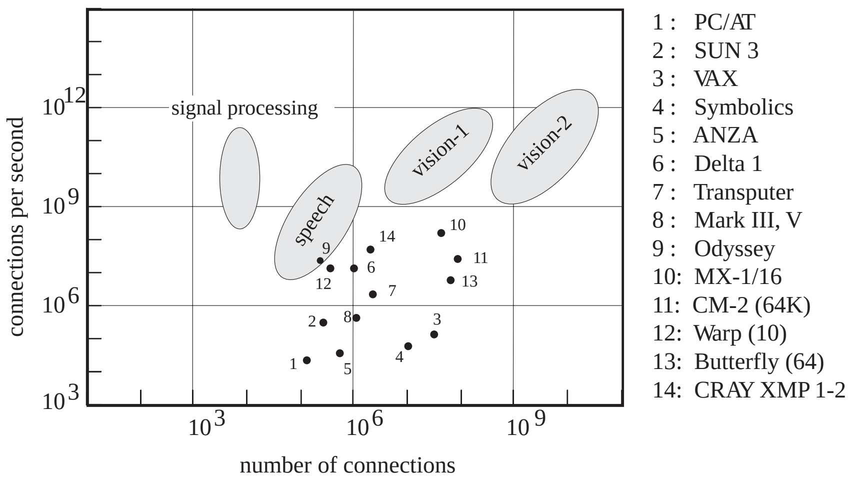


Fig. 18.1. Performance and capacity of different implementations of neural networks and performance requirements for some applications [Ramacher 1991]

Figure 18.1 shows that an old personal computer is capable of achieving up to 10Kcps, whereas a massively parallel CM-2 with 64K processors is capable of providing up to 10 Mcps of computing power. With such performance we can deal with speech recognition in its simpler form, but not with complex computer vision problems or more sophisticated speech recognition models. The performance necessary for this kind of application is of the order of Giga-cps. Conventional computers cannot offer such computing power with an affordable price-performance ratio. Neural network applications in which a person interacts with a computer require compact solutions, and this has been the motivation behind the development of many special coprocessors. Some of them are listed in Figure 18.1, such as the Mark III or ANZA boards [409].

A pure software solution is therefore not viable, yet it remains open whether an analog or a digital solution should be preferred. The next sections deal with this issue and discuss some of the systems that have been built, both in the analog and the digital world.

18.1.2 Types of neurocomputers

To begin with, we can classify the kinds of hardware solution that have been proposed by adopting a simple taxonomy of hardware models. This will simplify the discussion and help to get a deeper insight into the properties of each device. Defining a taxonomy of neurocomputers requires consideration of three important factors:

- the kind of signals used in the network,
- the implementation of the weights,
- the integration and output functions of the units.

The signals transmitted through the network can be coded using an analog or a digital model. In the analog approach, a signal is represented by the magnitude of a current, or a voltage difference. In the digital approach, discrete values are stored and transmitted. If the signals are represented by currents or voltages, it is straightforward to implement the weights using resistances or transistors with a linear response function for a certain range of values. In the case of a digital implementation, each transmission through one of the network's edges requires a digital multiplication.

These two kinds of network implementations indicate that we must differentiate between analog and digital neurocomputers. Figure 18.2 shows this first level of classification and some successive refinements of the taxonomy. Hybrid neurocomputers are built combining analog and digital circuits.

The analog approach offers two further alternatives: the circuits can be built using electronic or optical components. The latter alternative has been studied and has led to some working prototypes but not to commercial products, where electronic neurocomputers still dominate the scene.

In the case of a digital implementation, the first two subdivisions in Figure 18.2 refer to conventional parallel or pipelined architectures. The networks can be distributed in multiprocessor systems, or it can be arranged for the

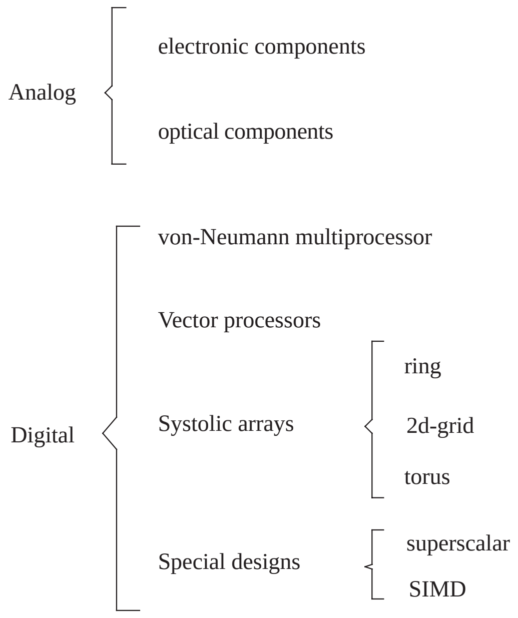


Fig. 18.2. Taxonomy of neurosystems

training set to be allocated so that each processor works with a fraction of the data. Neural networks can be efficiently implemented in vector processors, since most of the necessary operations are computations with matrices. Vector processors have been optimized for exactly such operations.

A third digital model is that of systolic arrays. They consist of regular arrays of processing units, which communicate only with their immediate neighbors. Input and output takes place at the boundaries of the array [207]. They were proposed to perform faster matrix-matrix multiplications, the kind of operation in which we are also interested for multilayered networks.

The fourth and last type of digital system consists of special chips of the superscalar type or systems containing many identical processors to be used in a SIMD (single instruction, multiple data) fashion. This means that all processors execute the same instruction but on different parts of the data.

Analog systems can offer a higher implementation density on silicon and require less power. But digital systems offer greater precision, programming flexibility, and the possibility of working with *virtual networks*, that is, networks which are not physically mapped to the hardware, making it possible to deal with more units. The hardware is loaded successively with different portions of the virtual network, or, if the networks are small, several networks can be processed simultaneously using some kind of multitasking.

18.2 Analog neural networks

In the analog implementation of neural networks a coding method is used in which signals are represented by currents or voltages. This allows us to think of these systems as operating with real numbers during the neural network simulation.

18.2.1 Coding

When signals are represented by currents, it is easy to compute their addition. It is only necessary to make them meet at a common point. One of the Kirchhoff laws states that the sum of all currents meeting at a point is zero (outgoing currents are assigned a negative sign). The simple circuit of Figure 18.3 can be used to add I_1 and I_2, and the result is the current I_3.

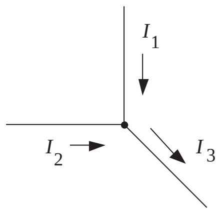


Fig. 18.3. Addition of electric current

The addition of voltage differences is somewhat more complicated. If two voltage differences V_1 and V_2 have to be added, the output line with voltage V_1 must be used as the reference line for voltage V_2. This simple principle cannot easily be implemented in real circuits using several different potentials $V_1, V_2, \ldots, V_n$. The representation of signals using currents is more advantageous in this case. The integration function of each unit can be implemented by connecting all incoming edges to a common point. The sum of the currents can be further processed by each unit.

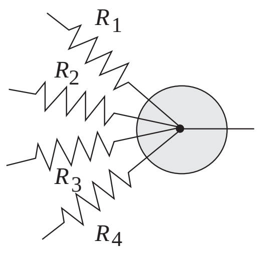


Fig. 18.4. Network of resistances

The weighting of the signal can be implemented using variable resistances. Rosenblatt used this approach in his first perceptron designs [185]. If the resistance is R and the current I, the potential difference V is given by Ohm's law $V = RI$. A network of resistances can simulate the necessary network connections and the resistors are the adaptive weights we need for learning (Figure 18.4).

Several analog designs for neural networks were developed in the 1970s and 1980s. Carver Mead's group at Caltech has studied different alternatives with which the size of the network and power consumption can be minimized [303]. In Mead's designs transistors play a privileged role, especially for the realization of so-called transconductance amplifiers. Some of the chips developed by Mead have become familiar names, like the *silicon retina* and the *silicon cochlea*.

Karl Steinbuch proposed at the end of the 1950s a model for associative learning which he called the *Lernmatrix* [414]. Figure 18.5 shows one of the columns of his learning matrix. It operated based on the principles discussed above.

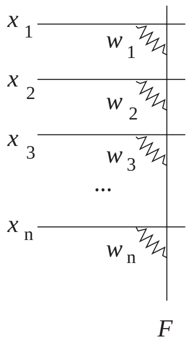


Fig. 18.5. A column of the *Lernmatrix*

The input $x_1, x_2, \ldots, x_n$ is transmitted as a voltage through each of the input lines. The resistances $w_1, w_2, \ldots, w_n$ transform the voltage into a weighted current. Several of these columns allow a vector-matrix product to be computed in parallel. Using such an array (and a special kind of nonlinearity) Steinbuch was able to build the first associative memory in hardware.

18.2.2 VLSI transistor circuits

Some VLSI circuits work with *field effect transistors* made of semiconductors. These are materials with a nonlinear voltage-current response curve. The nonlinear behavior makes them especially suitable for the implementation of digital switches.

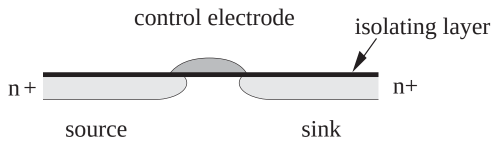


Fig. 18.6. Diagram of a field effect transistor (FET)

Figure 18.6 shows the structure of a typical field effect transistor. The source and the sink are made of n+ semiconductors, which contain a surplus of positive charge produced by carefully implanted impurities in the semiconductor crystal. A current can flow between source and sink only when the control electrode is positively charged above a certain threshold. Electrons are attracted from the source to the control electrode, but they also reach the sink by diffusing through the gap between source and sink. This closes the circuit and a current flows through the transistor. A potential difference is thus transformed into a current. The voltage difference is applied between source and control electrode, but the current that represents the signal flows between source and sink.

Let V_g be the potential at the control electrode, V_s the potential at the sink and V_{gs} the potential difference between control electrode and source. Assume further that V_{ds} is the potential difference between source and sink. It can be shown [303] that in this case the current I through the transistor is given by

$$I = I_0\, \mathrm{e}^{cV_g - V_s}(1 - \mathrm{e}^{V_{ds}}),$$

where I_0 and c are constants. The approximation $I = I_0\, \mathrm{e}^{cV_g - V_s}$ is valid for large enough negative V_{ds}. In this case the output current depends only on V_g and V_s.

It is possible to design a small circuit made of FETs capable of multiplying the numerical value of a voltage by the numerical value of a potential difference (Figure 18.7).

The three transistors T_1, T_2 and T_b are interconnected in such a way that the currents flowing through all of them meet at a common point. This means that $I_1 + I_2 = I_b$. But since

$$I_1 = I_0 \mathrm{e}^{cV_1 - V} \qquad \text{and} \qquad I_2 = I_0 \mathrm{e}^{cV_2 - V},$$

it is true that

$$I_b = I_1 + I_2 = I_0 \mathrm{e}^{-V}(\mathrm{e}^{cV_1} + \mathrm{e}^{cV2}).$$

Some additional algebraic steps lead to the following equation

$$I_1 - I_2 = I_b\, \frac{\mathrm{e}^{cV_1} - \mathrm{e}^{cV_2}}{\mathrm{e}^{cV_1} + \mathrm{e}^{cV_2}} = I_b \tanh \frac{c(V_1 - V_2)}{2}.$$

This equation shows that a potential difference $V_1 - V_2$ produces a nonlinear result $I_1 - I_2$. Using the circuit in Figure 18.7 we can implement the output

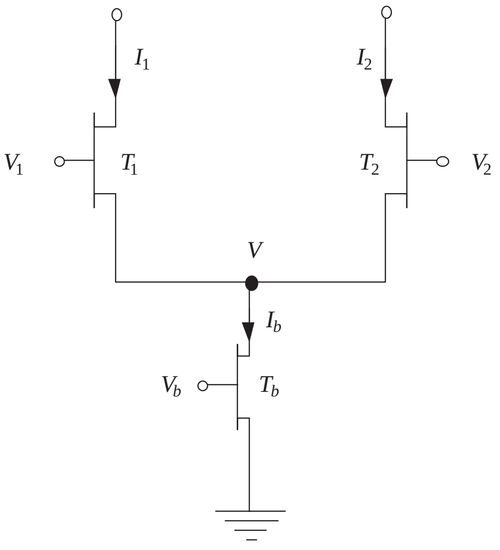


Fig. 18.7. Transconductance amplifier (Mead 1989)

function of a unit. Remember that the hyperbolic tangent is just the symmetric sigmoid used in backpropagation networks. For small values of x it holds that $\tanh(x) \approx x$. In this case for small values of $V_1 - V_2$ we get

$$I_1 - I_2 = \frac{cI_b}{2}(V_1 - V_2).$$

The circuit operates as a multiplier. The magnitude of the current I_b is multiplied by the magnitude of the potential difference $V_1 - V_2$, where we only require $I_b > 0$. An analog multiplier for two numbers with arbitrary sign requires more components and a more sophisticated design, but the basic construction principle is the same. An example is the Gilbert multiplier used in many VLSI circuits.

The above discussion illustrates how FETs can be used, and are in fact being used, to implement the main computations for hardware models of neural networks, that is, the weighting of signals, their integration, and the computation of the nonlinear output.

18.2.3 Transistors with stored charge

A possible alternative to the kind of components just discussed is to use *floating gate transistors* for an analog implementation of the multiplication operation. A problem with the circuits discussed above is that the numbers to be multiplied must be encoded as voltages and currents. The numerical value of the weights must be transformed into a potential difference, which must be kept constant as long as no weight update is performed. It would be simpler to store the weights using an analog method in such a way that they could be used when needed. This is exactly what floating gate transistors can achieve.

Figure 18.8 shows the scheme of such a transistor. The structure resembles that of a conventional field effect transistor. The main visible difference is the presence of a metallic layer between the control electrode and the silicon base of the transistor. This metallic layer is isolated from the rest of the transistor in such a way that it can store a charge just as a capacitor. The metallic layer can be charged by raising the potential difference between the control electrode and the source by the right amount. The charge stored in the metallic layer decays only slowly and can be stored for a relatively long time [185].

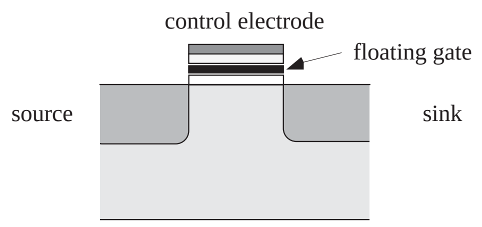


Fig. 18.8. Diagram of a floating gate transistor

A floating gate transistor can be considered as one which includes an analog memory. The stored charge modifies the effectiveness of the control electrode. The current which flows through source and sink depends linearly (in a certain interval) on the potential difference between them. The proportionality constant is determined by the stored charge. The potential difference is therefore weighted before being transformed into a current. Learning is very simple to implement: weight updates correspond to changes in the amount of charge stored. Even if the power source is disconnected the magnitudes of the weights remain unchanged.

18.2.4 CCD components

Floating gate transistors represent the weights by statically stored charges. A different approach consists in storing charges *dynamically*, as is done with the help of *charge coupled devices* (CCDs). Computation of the scalar product of a weight and an input vector can be arranged in such a way that the respective components are used sequentially. The weights, which are stored dynamically, can be kept moving in a closed loop. At a certain point in the loop, the magnitudes of the weights are read and multiplied by the input. The accumulated sum is the scalar product needed.

Figure 18.9 shows a CCD circuit capable of transporting charges. A chain of electrodes is built on top of a semiconductor base. Each electrode receives the signal of a clock. The clock pulses are synchronized in such a way that at every discrete time t just one of the electrodes reaches its maximum potential. At the next time step $t+1$ only the neighbor to the right reaches the maximum potential and so on. A charge Q is stored in the semiconductor material, which has been treated in order to create on its surface potential wells aligned with the electrodes. The charge stored in a potential well diffuses gradually (as shown for $t = 1.5$), but if the next electrode reaches its maximum before the charge has completely diffused, it is again collected as a charge packet in its potential well. One or more stored charge packets can move from left to right in this CCD circuit. If the chain of electrodes is arranged in a closed loop, the stored charge can circulate indefinitely.

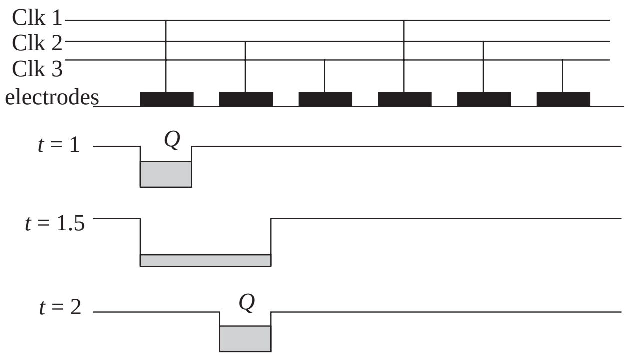


Fig. 18.9. CCD transport band

Figure 18.10 shows a very interesting design for an associative memory with discrete weights. The matrix of weights is stored in the n CCD linear transport arrays shown on the left. Each row of the weight matrix is stored in one of the arrays. At the end of the line a weight is read in every cycle and is transformed into a digital number using an A/D converter. The transport chain to the right stores the momentary states of the units (0 or 1). In each cycle a state is read and it is multiplied by each of the n weights from the CCD arrays to the left. Since the states and weights are single bits, an AND gate is enough. The partial products are kept in the accumulators for n cycles. At the end of this period of time the circuit has computed n scalar products which are stored in the n accumulators. A threshold function now converts the scalar products in the new unit states, which are then stored on the vertical CCD array.

Very compact arithmetic and logical units can be built using CCD technology. The circuit shown in Figure 18.10 is actually an example of a hybrid approach, since the information is handled using analog and digital coding. Although much of the computation is done serially, CCD components are fast enough. The time lost in the sequential computation is traded off against the

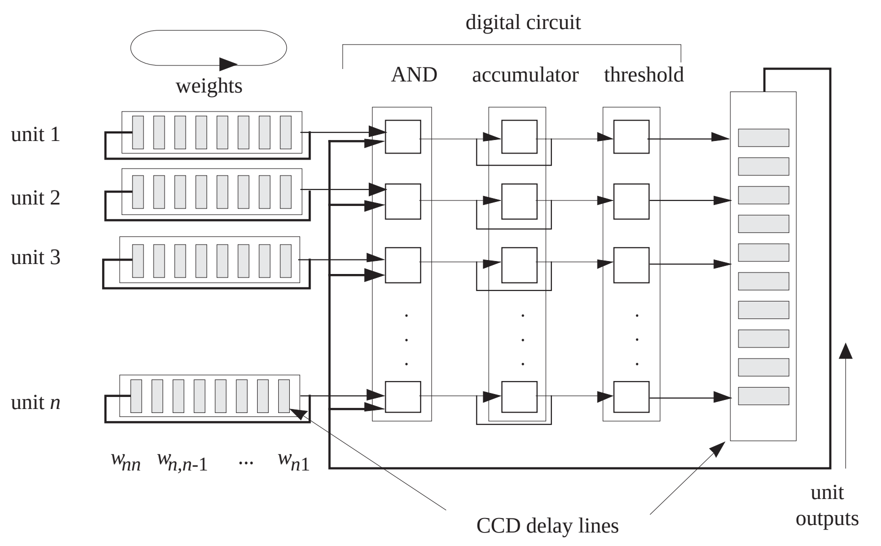


Fig. 18.10. Associative memory in CCD technology

simplicity of the design. The circuit works synchronously, because information is transported at discrete points in time.

18.3 Digital networks

In digital neurocomputers signals and network parameters are encoded and processed digitally. The circuits can be made to work with arbitrary precision just by increasing the word length of the machine. Analog neurocomputers are affected by the strong variation of the electrical characteristics of the transistors, even when they have been etched on the same chip [303]. According to the kind of analog component, the arithmetic precision is typically limited to 8 or 9 bits [185]. Analog designs can be used in all applications that can tolerate statistical deviations in the computation and that do not require more than the precision just mentioned, as is the case, for example, in computer vision. If we want to work with learning algorithms based on gradient descent, more precision is required.

There is another reason for the popularity of digital solutions in the neurocomputing field: our present computers work digitally and in many cases the neural network is just one of the pieces of a whole application. Having everything in the same computer makes the integration of the software easier. This has led to the development of several boards and small machines connected to a host as a neural coprocessor.

18.3.1 Numerical representation of weights and signals

If a neural network is simulated in a digital computer, the first issue to be decided is how many bits should be used for storage of the weights and signals. Pure software implementations use some kind of floating-point representation, such as the IEEE format, which provides very good numerical resolution at the cost of a high investment in hardware (for example for the 64-bit representations). However, floating-point operations require more cycles to be computed than their integer counterparts (unless very complex designs are used). This has led most neurocomputer designers to consider fixed-point representations in which only integers are used and the position of the decimal point is managed by the software or simple additional circuits. If such a representation is used, the appropriate word length must be found. It must not affect the convergence of the learning algorithms and must provide enough resolution during normal operation. The classification capabilities of the trained networks depend on the length of the bit representation [223].

Some researchers have conducted series of experiments to find the appropriate word length and have found that 16 bits are needed to represent the weights and 8 to represent the signals. This choice does not affect the convergence of the backpropagation algorithm [31, 197]. Based on these results, the design group at Oregon decided to build the CNAPS neurocomputer [175] using word lengths of 8 and 16 bits (see Sect. 8.2.3).

Experience with some of the neurocomputers commercially available shows that the 8- and 16-bit representations provide enough resolution in most, but not in all cases. Furthermore, some complex learning algorithms, such as variants of the conjugate gradient methods, require high accuracy for some of the intermediate steps and cannot be implemented using just 16 bits [341]. The solution found by the commercial suppliers of neurocomputers should thus be interpreted as a compromise, not as the universal solution for the encoding problem in the field of neurocomputing.

18.3.2 Vector and signal processors

Almost all models of neural networks discussed in the previous chapters require computation of the scalar product of a weight and an input vector. Vector and signal processors have been built with this type of application in mind and are therefore also applicable for neurocomputing.

A vector processor, such as a CRAY, contains not only scalar registers but also vector registers, in which complete vectors can be stored. To perform operations on them, they are read sequentially from the vector registers and their components are fed one after the other to the arithmetic units. It is characteristic for vector processors that the arithmetic units consist of several stages connected in a pipeline. The multiplier in Figure 18.11, for example, performs a single multiplication in 10 cycles. If the operand pairs are fed sequentially into the multiplier, one pair in each cycle, at the end of cycle 10 one result has been computed, another at the end of cycle 11, etc. After a start time of 9 cycles a result is produced in every cycle. The partial results can be accumulated with an adder. After $9 + n$ cycles the scalar product has been computed.

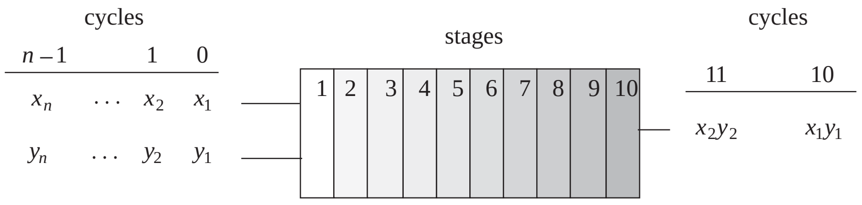


Fig. 18.11. Multiplier with 10 pipeline sections

The principle of vector processors has been adopted in signal processors, which always include fast pipelined multipliers and adders. A multiplication can be computed simultaneously with an addition. They differ from vector processors in that no vector registers are available and the vectors must be transmitted from external memory modules. The bottleneck is the time needed for each memory access, since processors have become extremely fast compared to RAM chips [187]. Some commercial neurocomputers are based on fast signal processors coupled to fast memory components.

18.3.3 Systolic arrays

An even greater speedup of the linear algebraic operations can be achieved with *systolic arrays*. These are regular structures of VLSI units, mainly one- or two-dimensional, which can communicate only locally. Information is fed at the boundaries of the array and is transported synchronously from one stage to the next. Kung gave these structures the name systolic arrays because of their similarity to the blood flow in the human heart. Systolic arrays can compute the vector-matrix multiplication using fewer cycles than a vector processor. The product of an n-dimensional vector and an $n \times n$ matrix can be computed in $2n$ cycles. A vector processor would require in the order of n^2 cycles for the same computation.

Figure 18.12 shows an example of a two-dimensional systolic array capable of computing a vector-matrix product. The rows of the matrix

$$\mathbf{W} = \begin{pmatrix} c & d \\ e & f \end{pmatrix}$$

are used as the input from the top into the array. In the first cycle c and d constitute the input. In the second cycle e and f are fed into the array, but displaced by one unit to the right. We want to multiply the vector (a, b)

with the matrix $\mathbf{W}$. The vector (a, b) is fed into the array from the left. Each unit in the systolic array multiplies the data received from the left and from above and the result is added to the data transmitted through the diagonal link from the neighbor located to the upper left. Figure 18.12 shows which data is transmitted through which links (for different cycles). After two cycles, the result of the vector-matrix multiplication is stored in the two lower units to the right. Exactly the same approach is used in larger systolic arrays to multiply n-dimensional vectors and $n \times n$ matrices.

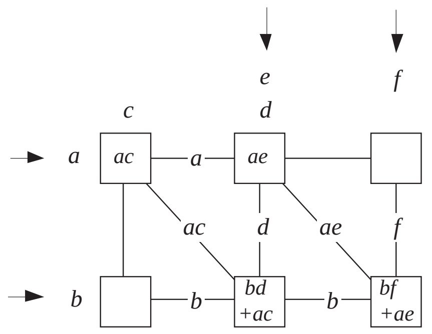


Fig. 18.12. Planar systolic array for vector-matrix multiplication

The systolic array used in the above example can be improved by connecting the borders of the array to form a torus. The previous vector-matrix multiplication is computed in this case, as shown in Figure 18.13. In general it holds that in order to multiply an n-dimensional vector with an $n \times n$ matrix, a systolic array with $n(2n-1)$ elements is needed. A torus with $n \times n$ elements can perform the same computation. Therefore when systolic arrays are used for neurocomputers, a toroidal architecture is frequently selected.

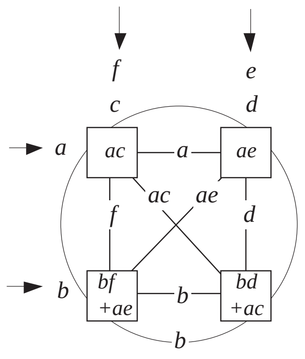


Fig. 18.13. Systolic array with toroidal topology

An interesting systolic design is the one developed by Siemens (Figure 18.14). It consists of a two-dimensional array of multipliers. The links for the transmission of weights and data are 16 bits wide. The array shown in the figure can compute the product of four weights and four inputs. Each multiplier gets two arguments which are then multiplied. The result is transmitted to an adder further below in the chain. The last adder eventually contains the result of the product of the four-dimensional input with the four-dimensional weight vector. The hardware implementation does not exactly correspond to our diagram, since four links for the inputs and four links for the weights are not provided. Just one link is available for the inputs and one for the weights, but both links operate by multiplexing the input data and the weights, that is, they are transmitted sequentially in four time frames. Additional hardware in the chip stores each weight as needed for the multiplication and the same is done for the inputs. In this way the number of pins in the final chip could be dramatically reduced without affecting performance [354]. This kind of multiplexed architecture is relatively common in neurocomputers [38].

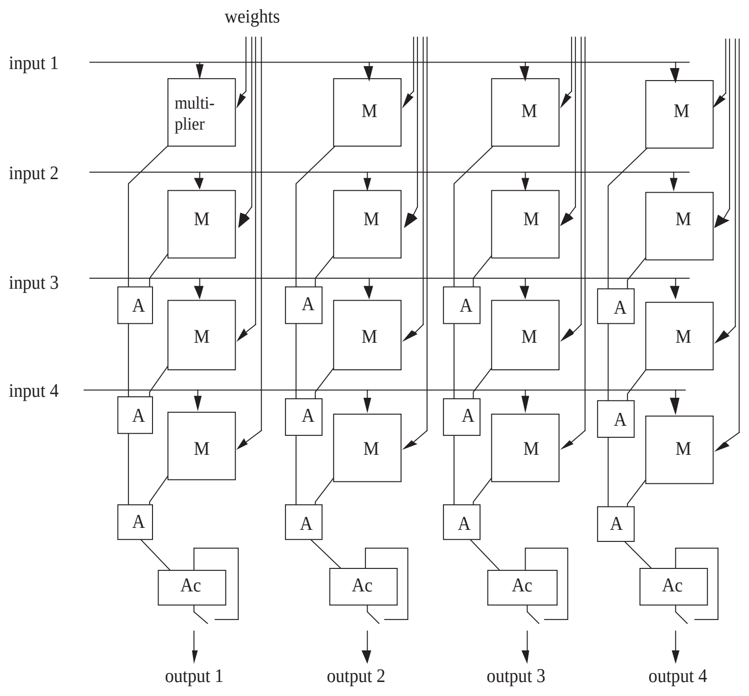


Fig. 18.14. Systolic array for 4 scalar product chains

The systolic chips manufactured by Siemens (MA16) can be arranged in even larger two-dimensional arrays. Each MA16 chip is provided with an external memory in order to store some of the network's weights. The performance promised by the creators of the MA16 should reach 128 Gcps for a 16×16 systolic array [354]. This kind of performance could be used, for example, in computer vision tasks. Siemens started marketing prototypes of the *Synapse* neurocomputer in 1995.

18.3.4 One-dimensional structures

The hardware invested in two-dimensional systolic arrays is in many cases excessive for the vector-matrix multiplication. This is why some researchers have proposed using one-dimensional systolic arrays in order to reduce the complexity of the hardware without losing too much performance [438, 263].

In a systolic ring information is transmitted from processor to processor in a closed loop. Each processor in the ring simulates one of the units of the neural network. Figure 18.15 shows a ring with four processors. Assume that we want to compute the product of the 4×4 weight matrix $\mathbf{W} = \{w_{ij}\}$ with the vector $x^{\mathrm{T}} = (x_1, x_2, \ldots, x_n)$, that is, the vector $\mathbf{Wx}$. The vector $\mathbf{x}$ is loaded in the ring as shown in the figure. In each cycle each processor accesses a weight from its local memory (in the order shown in Figure 18.15) and gets a number from its left neighbor. The weight is multiplied by x_i and the product is added to the number received from the left neighbor. The number from the left neighbor is the result of the previous multiply-accumulate operation (the ring is initialized with zeros). The reader can readily verify that after four cycles (a complete loop) the four small processors contain the four components of the vector $\mathbf{Wx}$.

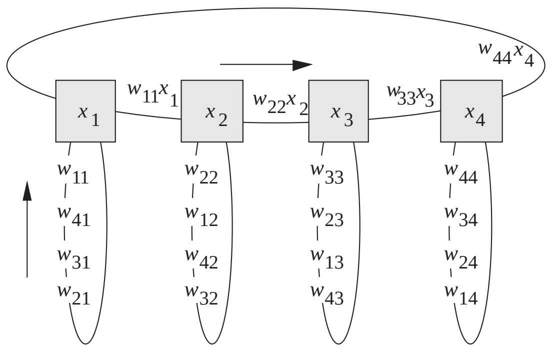


Fig. 18.15. Systolic array computing the product of $\mathbf{W}$ and $\mathbf{x}$

It is not necessary for the units in the ring to be fully-fledged processors. They can be arithmetic units capable of multiplying and adding and which contain some extra registers. Such a ring can be expanded to work with larger matrices just by including additional nodes.

For the backpropagation algorithm it is also necessary to multiply vectors with the transpose of the weight matrix. This can be implemented in the

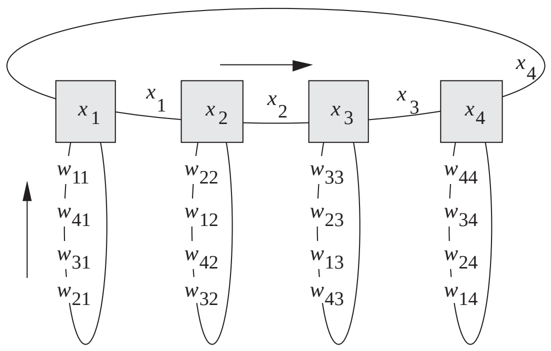

x_1
x_2
x_3
x_4
x_1
x_2
x_3
x_4
w_{11} w_{41} w_{31} w_{21}
w_{22} w_{12} w_{42} w_{32}
w_{33} w_{23} w_{13} w_{43}
w_{44} w_{34} w_{24} w_{14}

Fig. 18.16. Systolic array computing the product of the transpose **W** and **x**

same ring by making some changes. In each cycle the number transmitted to the right is x_i and the results of the multiplication are accumulated at each processor. After four cycles the components of the product $\mathbf{W}^{\mathrm{T}}\mathbf{x}$ are stored at each of the four units.

A similar architecture was used in the Warp machine designed by Pomerleau and others at Princeton [347]. The performance reported was 17 Mcps. Jones et al. have studied similar systolic arrays [226].

Another machine of a systolic type is the RAP (Ring Array Processor) developed at the International Computer Science Institute in Berkeley. The processing units consist of signal processors with a large local memory [321]. The weights of the network are stored at the nodes in the local memories. The performance reported is 200–570 Mcps using 10 processor nodes in a ring. It is significant that the RAP has been in constant use for the last five years at the time of this writing. It seems to be the neural computer with the largest accumulated running time.

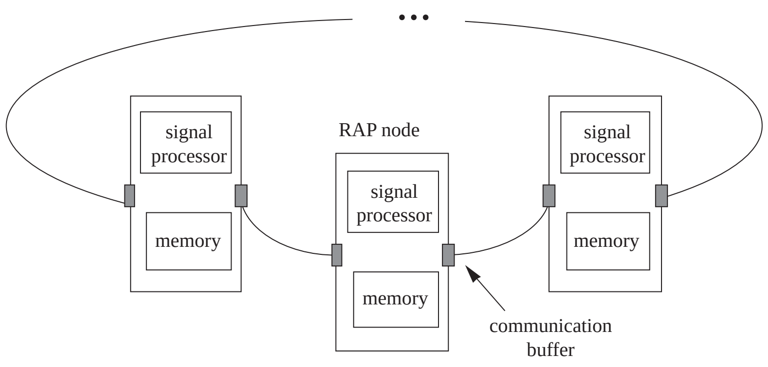


Fig. 18.17. Architecture of the Ring Array Processor

Figure 18.17 shows the architecture of the RAP. The local memory of each node is used to store a row of the weight matrix $\mathbf{W}$, so that each processor can compute the scalar product of a row with the vector $(x_1, x_2, \ldots, x_n)^{\mathrm{T}}$. The input vector is transmitted from node to node. To compute the product $\mathbf{W}^{\mathrm{T}}\mathbf{x}$ partial products are transmitted from node to node in systolic fashion. The weight updates can be made locally at each node. If the number of processors is lower than the dimension of the vector and matrix rows, then each processor stores two or more rows of the weight matrix and computes as much as needed. The advantage of this architecture compared to a pure systolic array is that it can be implemented using off-the-shelf components. The computing nodes of the RAP can also be used to implement many other non-neural operations needed in most applications.

18.4 Innovative computer architectures

The experience gained from implementations of neural networks in conventional von Neumann machines has led to the development of complete microprocessors especially designed for neural networks applications. In some cases the SIMD computing model has been adopted, since it is relatively easy to accelerate vector-matrix multiplications significantly with a low hardware investment. Many authors have studied optimal mappings of neural network models onto SIMD machines [275].

18.4.1 VLSI microprocessors for neural networks

In SIMD machines the same instruction is executed by many processors using different data. Figure 18.18 shows a SIMD implementation of a matrix-vector product. A chain of n multipliers is coupled to n adders. The components of a vector $x_1, x_2, \ldots, x_n$ are transmitted sequentially to all the processors. At each cycle one of the columns of the matrix $\mathbf{W}$ is transmitted to the processors. The results of the n multiplications are accumulated in the adders and at the same time the multipliers begin to compute the next products. For such a computation model to work efficiently it is necessary to transmit n arguments from the memory to the arithmetic units at each cycle. This requires a very wide instruction format and a wide bus.

The Torrent chip completed and tested at Berkeley in the course of 1995 has a SIMD architecture in each of its vector pipelines [440] and was the first vector processor on a single chip. The chip was designed for neural network but also for other digital signal processing tasks. It consists of a scalar unit compatible with a MIPS-II 32-bit integer CPU. A fixed point coprocessor is also provided.

Figure 18.19 shows a block diagram of the chip. It contains a register file for 16 vectors, each capable of holding 32 elements, each of 32 bits. The pipeline of the chip is superscalar. An instruction can be assigned to the CPU, the

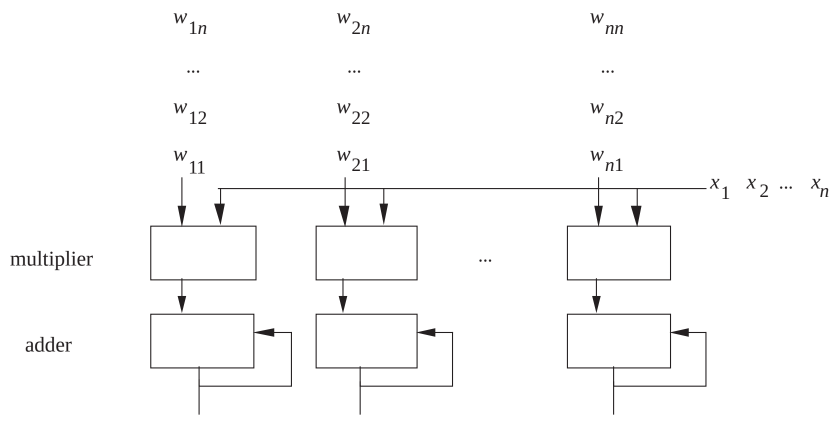


Fig. 18.18. SIMD model for vector-matrix multiplication

Vector memory unit (VMP) or any of two vector processing units (VP0 and VP1). The vector units can add, shift, and do conditional moves. Only VP0 can multiply. The multipliers in VP0 perform $16 \times 16 \rightarrow 32$ bit multiplications. The vector memory operations provided allow efficient loading and storage of vectors.

Interesting is that all three vector functional units are composed of 8 parallel pipelines. This means that the vector is striped along the 8 pipelines. When all three vector units are saturated, up to 24 operations per cycle are being executed.

Due to its compatibility with the MIPS series, the scalar software can be compiled with the standard available tools. The reported performance with 45 MHz is 720 Mflops [33].

In the Torrent the weight matrix is stored outside the VLSI chip. In the CNAPS (Connected Network of Adaptive Processors) built and marketed by Adaptive Solutions, the weights are stored in local memories located in the main chip [175]. The CNAPS also processes data in SIMD fashion. Each CNAPS chip contains 64 processor nodes together with their local memory.

Figure 18.20 shows the basic structure of the CNAPS chips, which have been built using Ultra-VLSI technology: more than 11 million transistors have been etched on each chip. Every one of them contains 80 processor nodes, but after testing only 64 are retained to be connected in a chain [176]. The 64 nodes receive the same input through two buses: the instruction bus, necessary to coordinate the 64 processing units, and the input bus. This has a width of only 8 bits and transmits the outputs of the simulated units to the other units. The weights are stored locally at each node. The programmer can define any computation strategy he or she likes, but usually what works best is to store all incoming weights in a unit in the local memory, as well as all outgoing weights (necessary for the backpropagation algorithm). The output bus, finally, communicates the output of each unit to the other units.

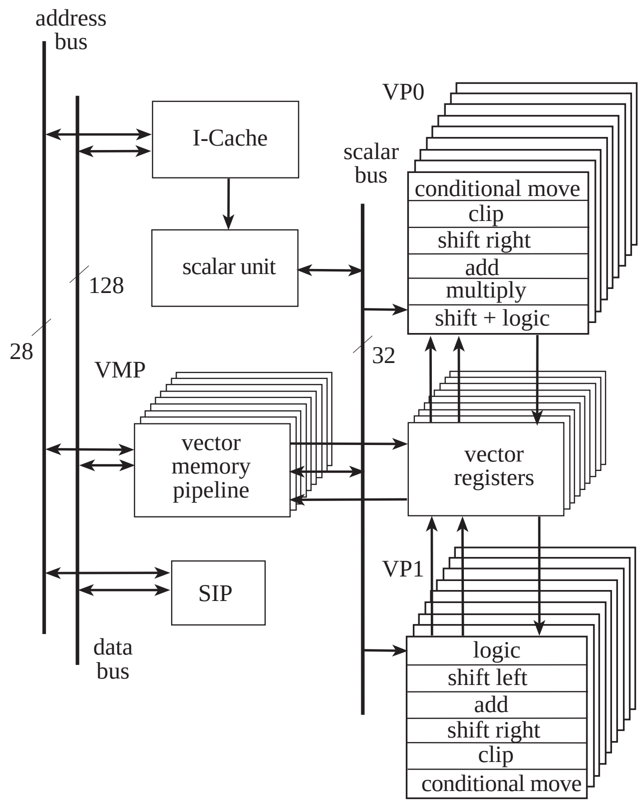


Fig. 18.19. Torrent block diagram [Asanovic et al. 95]

Figure 18.21 shows a block diagram of a processor node. A multiplier, an adder, and additional logical functions are provided at each node. The register file consists of 32 16-bit registers. The local memory has a capacity of 4Kb. The structure of each node resembles a RISC processor with additional storage. Using one of the chips with 64 processing nodes it is possible to multiply an 8-bit input with 64 16-bit numbers in parallel. Several CNAPS chips can be connected in a linear array, so that the degree of parallelism can be increased in quanta of 64 units. As in any other SIMD processor it is possible to test status flags and switch-off some of the processors in the chain according to the result of the test. In this way an instruction can be executed by a subset of the processors in the machine. This increases the flexibility of the system. A coprocessor with 4 CNAPS chips is capable of achieving 5 Gcps and 944 Mcups.

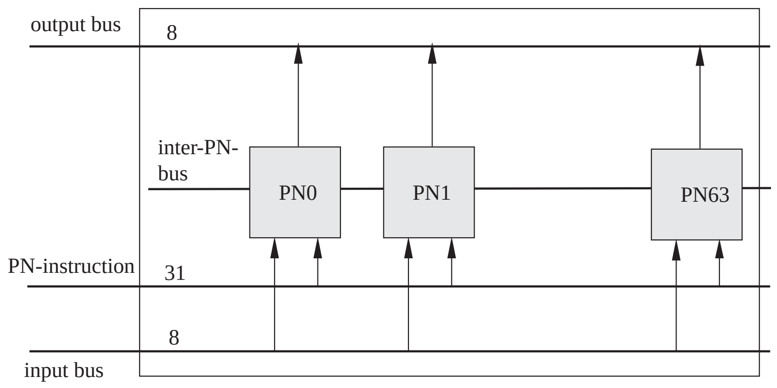


Fig. 18.20. CNAPS-Chip with 64 processor nodes (PN)

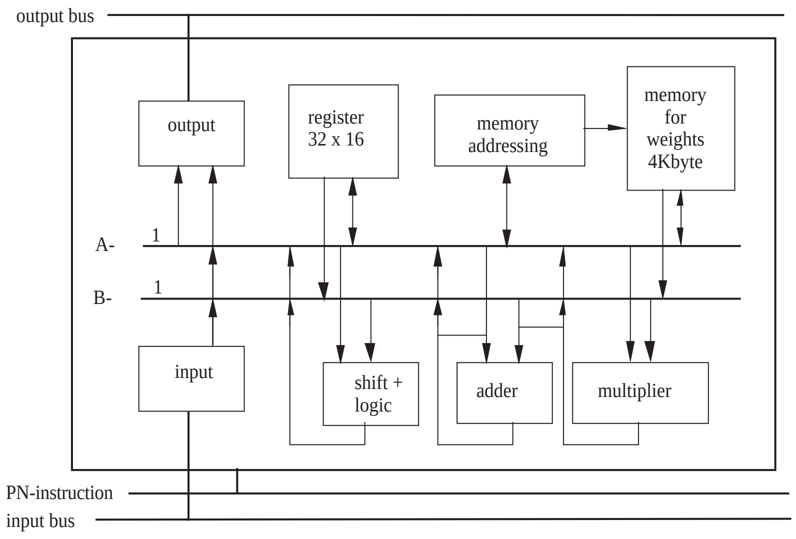


Fig. 18.21. Structure of a processor node

18.4.2 Optical computers

The main technical difficulty that has to be surmounted in neural networks, and other kinds of massively parallel systems, is the implementation of communication channels. A Hopfield network with 1000 units contains half a million connections. With these large topologies we have no alternative but to work with virtual networks which are partitioned to fit on the hardware at hand. This is the approach followed for the digital implementations discussed in the last section. Optical computers have the great advantage, compared to electronic machines, that the communication channels do not need to be hard-wired. Signals can be transmitted as light waves from one component to the other. Also, light rays can cross each other and this does not affect the information they are carrying. The energy needed to transmit signals is low, since there is no need to consider the capacity of the transmitting medium, as in the case of metal cables. Switching times of up to 30 GHz can be achieved with optical elements [296].

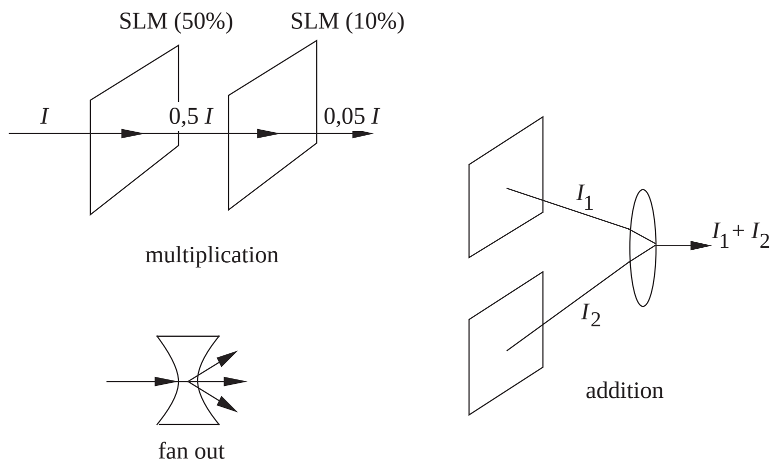


Fig. 18.22. Optical implementation of multiplication, addition, and signal splitting

Using optical signals it is possible to implement basic logic operations in a straightforward way. To do this, SLMs (Spatial Light Modulators) are used. These are optical masks which can be controlled using electrodes. According to the voltage applied the SLM becomes darker and reduces the amount of transmitted light when a light ray is projected onto the mask. A light signal projected onto an SLM configured to let only 50% of the light through loses half its intensity. This can be interpreted as multiplication of the number 1 by the number 0.5. A chain of multiplications can be computed almost "instantly" by letting the light ray go through several SLMs arranged in a stack. The light coming out has an intensity proportional to the product of the darkness ratios of the SLMs.

Addition of optical signals can be implemented by reducing two light signals to a single one. This can be done using lenses, prisms, or any other devices capable of dealing with light rays. A signal can be split in two or more using crystals or certain lens types. Figure 18.22 shows possible realizations of some operations necessary to implement optical computers.

Using these techniques all linear algebraic operations can be computed in parallel. To perform a vector-matrix or a matrix-vector multiplication it is only necessary to use an SLM that has been further subdivided into $n \times n$ fields. The darkness of each field must be adjustable, although in some applications it is

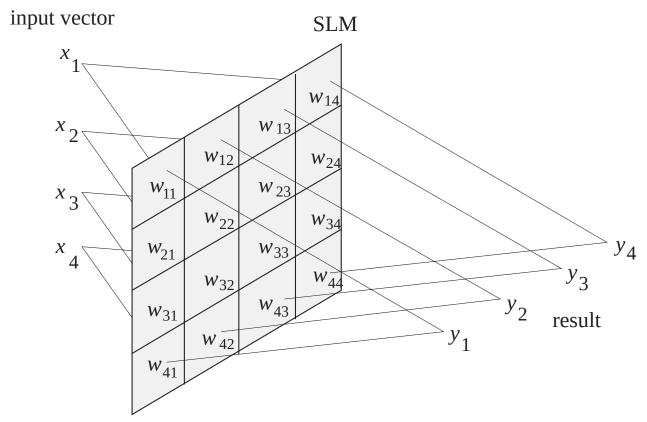


Fig. 18.23. Matrix-vector multiplication with an SLM mask

possible to use constant SLM masks. Each field represents one of the elements of the weight matrix and its darkness is adjusted according to its numerical value. The vector to be multiplied with the weight matrix is projected onto the SLM in such a way that the signal x_1 is projected onto the first row of the SLM matrix, x_2 onto the second row and so on. The outcoming light is collected column by column. The results are the components of the product we wanted to compute.

Fig. 18.24. Optical implementation of the backpropagation algorithm

Many other important operations can be implemented very easily using optical computers [134]. Using special lenses, for example, it is possible to instantly compute the Fourier transform of an image. Some pattern recognition problems can be solved more easily taking the image from the spatial to the frequency domain (see Chap. 12). Another example is that of feedback systems of the kind used for associative memories. Cellular automata can be also implemented using optical technology. Figure 18.24 shows a diagram of an experiment in which the backpropagation algorithm was implemented with optical components [296].

The diagram makes the main problem of today's optical computers evident: they are still too bulky and are mainly laboratory prototypes still waiting to be miniaturized. This could happen if new VLSI techniques were developed and new materials discovered that could be used to combine optical with electronic elements on the same chips. However, it must be kept in mind that just 40 years ago the first transistor circuits were as bulky as today's optical devices.

18.4.3 Pulse coded networks

All of the networks considered above work by transmitting signals encoded analogically or digitally. Another approach is to implement a closer simulation of the biological model by transmitting signals as discrete pulses, as if they were action potentials. Biological systems convey information from one neuron to another by varying the firing rate, that is, by something similar to frequency modulation. A strong signal is represented by pulses produced with a high frequency. A feeble signal is represented by pulses fired with a much lower frequency. It is not difficult to implement such pulse coding systems in analog or digital technology.

Tomlinson et al. developed a system which works with discrete pulses [430]. The neurochip they built makes use of an efficient method of representing weights and signals. Figure 18.25 shows an example of the coding of two signals using asynchronous pulses.

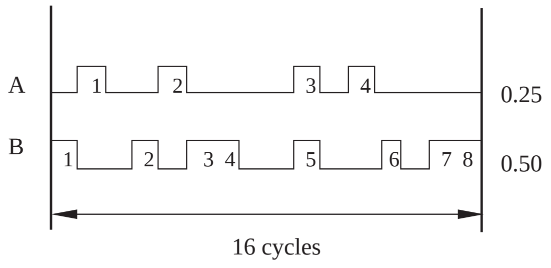


Fig. 18.25. Pulse coded representation of signals

Assume that two signals A and B have the respective numerical values 0.25 and 0.5. All signals are represented by pulses in a 16-cycles interval. A generator produces square pulses randomly, but in such a way that for signal A, a pulse appears in the transmitted stream a quarter of the time. In the case of B, a pulse is produced half of the time. The pulse trains are uncorrelated to each other. This is necessary in order to implement the rest of the logic functions. A decoder can reconstruct the original numbers just by counting the number of pulses in the 16-cycle interval.

The product of the two numbers A and B can be computed using a single AND gate. As shown in Figure 18.26, the two pulse streams are used as the input to the AND gate. The result is a pulse chain containing only $0.25 \times 0.5 \times 16$ pulses. This corresponds to the number 0.25×0.5, that is, the product of the two arguments A and B. This is of course only a statistical result, since the number of pulses can differ from the expected average, but the accuracy of the computation can be increased arbitrarily by extending the length of the coding interval. Tomlinson found in his experiments that 256 cycles were good enough for most of the applications they considered. This corresponds, more or less, to a signal resolution of 8 bits.

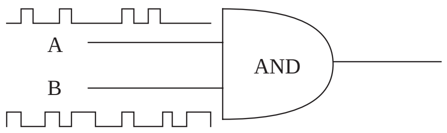


Fig. 18.26. Multiplication of two pulse coded numbers

The integration and nonlinear output function of a unit can be computed using an OR gate. If two numbers A and B are coded as indicated and are used as arguments, the result is a train of pulses which corresponds to the number $C = 1 - (1 - A)(1 - B)$. This means that we always get a one as the result, except in the case where both pulse trains contain zeroes. This happens with probability $(1 - A)(1 - B)$. The result C corresponds to a kind of summation with an upper saturation bound which restricts the output to the interval $[0, 1]$. Ten pulse-coded numbers $A_1, A_2, \ldots, A_{10}$ can be integrated using an OR gate. For small values of A_i the result is

$$C = 1 - (1 - A_1) \cdots (1 - A_{10}) \approx 1 - (1 - \sum_{i=1}^{10} A_i) \approx \sum_{i=1}^{10} A_i.$$

It can be shown that for larger magnitudes of the signal and a wide enough coding interval, the approximation

$$C = 1 - \exp(-\sum_{i=1}^{10} A_i)$$

holds. This function has the shape of a squashing function similar to the sigmoid.

The kind of coding used does not allow us to combine negative and positive signals. They must be treated separately. Only when the activation of a unit has to be computed do we need to combine both types of signals. This requires additional hardware, a problem which also arises in other architectures, for example in optical computers.

How to implement the classical learning algorithms or their variants using pulse coding elements has been intensively studied . Other authors have built analog systems which implement an even closer approximation to the biological model, as done for example in [49].

18.5 Historical and bibliographical remarks

The first attempts to build special hardware for artificial neural networks go back to the 1950s in the USA. Marvin Minsky built a system in 1951 that simulated adaptive weights using potentiometers [309]. The perceptron machines built by Rosenblatt are better known. He built them from 1957 to 1958 using Minsky's approach of representing weights by resistances in an electric network.

Rosenblatt's machines could solve simple pattern recognition tasks. They were also the first commercial neurocomputers, as we call such special hardware today. Bernard Widrow and Marcian Hoff developed the first series of adaptive systems specialized for signal processing in 1960 [450]. They used a special kind of vacuum tube which they called a *memistor*. The European pioneers were represented by Karl Steinbuch, who built associative memories using resistance networks [415].

In the 1970s there were no especially important hardware developments for neural networks, but some attempts were made in Japan to actually build Fukushima's cognitron and neocognitron [144, 145].

Much effort was invested in the 1980s to adapt conventional multiprocessor systems to the necessities of neural networks. Hecht-Nielsen built the series of Mark machines, first using conventional microprocessors and later by developing special chips. Some other researchers have done a lot of work simulating neural networks in vector computers or massively parallel systems.

The two main fields of hardware development were clearly defined in the middle of the 1980s. In the analog world the designs of Carver Mead and his group set the stage for further developments. In the digital world many alternatives were blooming at this time. Zurada gives a more extensive description of the different hardware implementations of neural networks [469].

Systolic arrays were developed by H. T. Kung at Carnegie Mellon in the 1970s [263]. The first systolic architecture for neural networks was the Warp machine built at Princeton. The RAP and Synapse machines, which are not purely systolic designs, nevertheless took their inspiration from the systolic model.

But we are still awaiting the greatest breakthrough of all: when will optical computers become a reality? This is the classical case of the application, the neural networks, waiting for the machine that can transform all their promises into reality.

Exercises

1. Train a neural network using floating-point numbers with a limited precision for the mantissa, for example 12 bits. This can be implemented easily by truncating the results of arithmetic operations. Does backpropagation converge? What about other fast variations of backpropagation?
2. Show how to multiply two $n \times n$ matrices using a two-dimensional systolic array. How many cycles are needed? How many multipliers?
3. Write the pseudocode for the backpropagation algorithm for the CNAPS. Assume that each processor node is used to compute the output of a single unit. The weights are stored in the local memory of the PNs.
4. Propose an optical system of the type shown in Figure 18.23, capable of multiplying a vector with a matrix $\mathbf{W}$, represented by an SLM, and also with its transpose.

.

References

1. Aarts, E., and J. Korst (1989), *Simulated Annealing and Boltzmann Machines*, John Wiley, Chichester, UK.
2. Abu-Mostafa, Y., and J. St. Jacques (1985), “Information Capacity of the Hopfield Model”, *IEEE Transactions on Information Theory*, Vol. IT–31, No. 4, pp. 461–464.
3. Abu-Mostafa, Y. (1986), “Neural Networks for Computing”?, in: [Denker 1986], pp. 1–6.
4. Abu-Mostafa, Y. (1989), “The Vapnik-Chervonenkis Dimension: Information Versus Complexity in Learning”, *Neural Computation*, Vol. 1, pp. 312–317.
5. Abu-Mostafa, Y. (1990), “Learning from Hints in Neural Networks”, *Journal of Complexity*, Vol. 6, pp. 192–198.
6. Abu-Mostafa, Y. (1993), “Hints and the VC Dimension”, *Neural Computation*, Vol. 5, No. 2, pp. 278–288.
7. Ackley, D., G. Hinton, and T. Sejnowski (1985), “A Learning Algorithm for Boltzmann Machines”, *Cognitive Science*, Vol. 9, pp. 147–169.
8. Ackley, D. (1987a), *A Connectionist Machine for Genetic Hillclimbing*, Kluwer, Boston, MA.
9. Ackley, D. (1987b), “An Empirical Study of Bit Vector Function Optimization”, in: [Davis 1987], pp. 170–204.
10. Agmon, S. (1954), “The Relaxation Method for Linear Inequalities”, *Canadian Journal of Mathematics*, Vol. 6, No. 3, pp. 382–392.
11. Aho, A., J. Hopcroft, and J. Ullman (1974), *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA.
12. Ainsworth, W. (1988), *Speech Recognition by Machine*, Peter Peregrinus, London.
13. Akiyama, Y., A. Yamashita, M. Kajiura, and H. Aiso (1989), “Combinatorial Optimization with Gaussian Machines”, in: [IEEE 1989], Vol. I, pp. 533–540.
14. Albert, A. (1972), *Regression and the Moore-Penrose Pseudoinverse*, Academic Press, New York.
15. Albrecht, R. F., C. R. Reeves, and N. C. Steele (eds.) (1993), *Artificial Neural Nets and Genetic Algorithms*, Springer-Verlag, Vienna.
16. Aleksander, I., and H. Morton (1990), *An Introduction to Neural Computing*, Chapman and Hall, London.

17. Aleksander, I. (1991), "Connectionism or Weightless Neurocomputing"?, in: [Kohonen et al. 1991], pp. 991–1000.
18. Aleksander, I., and J. Taylor (eds.) (1992), *Artificial Neural Networks 2*, Elsevier Science Publishers, Amsterdam.
19. Alexandrow, P. (ed.) (1983), *Die Hilbertschen Probleme*, Akademische Verlagsgesellschaft, Leipzig.
20. Almeida, L. B. (1987), "A Learning Rule for Asynchronous Perceptrons with Feedback in a Combinatorial Environment", in: [IEEE 1987], Vol. II, pp. 609–618.
21. Alon, N., and J. Bruck (1991), "Explicit Construction of Depth–2 Majority Circuits for Comparison and Addition", IBM Tech. Report RJ8300, San Jose, CA.
22. Amaldi, E. (1991), "On the Complexity of Training Perceptrons", in: [Kohonen et al. 1991], pp. 55–60.
23. Amari, S. (1977), "Neural Theory of Association and Concept Formation", *Biological Cybernetics*, Vol. 26, pp. 175–185.
24. Amit, D., H. Gutfreund, and H. Sompolinsky (1985), "Storing Infinite Numbers of Patterns in a Spin-Glass Model of Neural Networks", *Physical Review Letters*, Vol. 55, No. 14, pp. 1530–1533.
25. Amit, D. (1989), *Modeling Brain Function: The World of Attractor Neural Networks*, Cambridge University Press, Cambridge, UK.
26. Anderson, J., and E. Rosenfeld (1988), *Neurocomputing: Foundations of Research*, MIT Press, Cambridge, MA.
27. Arbib, M. (1987), *Brains, Machines and Mathematics*, Springer-Verlag, New York.
28. Arbib, M. (ed.) (1995), *The Handbook of Brain Theory and Neural Networks*, MIT Press, Cambridge, MA.
29. Armstrong, M. (1983), *Basic Topology*, Springer-Verlag, Berlin.
30. Arrowsmith, D., and C. Place (1990), *An Introduction to Dynamical Systems*, Cambridge University Press, Cambridge, UK.
31. Asanovic, K., and N. Morgan (1991), "Experimental Determination of Precision Requirements for Back-Propagation Training of Artificial Neural Networks", International Computer Science Institute, Technical Report TR–91-036, Berkeley, CA.
32. Asanovic, K., J. Beck, B. Kingsbury, P. Kohn, N. Morgan, and J. Wawrzynek (1992), "SPERT: A VLIW/SIMD Microprocessor for Artificial Neural Network Computations", International Computer Science Institute, Technical Report, TR–91-072, Berkeley, CA.
33. Asanovic, K., J. Beck, B. Irissou, B. Kingsbury, N. Morgan and J. Wawrzynek (1995), "The T0 Vector Microprocessor", *Hot Chips VII*, Stanford University, August.
34. Ashton, W. (1972), *The Logit Transformation*, Charles Griffin & Co., London.
35. Axelrod, R., and W. Hamilton (1981), "The Evolution of Cooperation", *Science*, Vol. 211, pp. 1390–1396.
36. Axelrod, R. (1987), "The Evolution of Strategies in the Iterated Prisoner's Dilemma", in: [Davis 1987], pp. 32–41.
37. Bäck, T., F. Hoffmeister, and H. P. Schwefel (1991), "A Survey of Evolution Strategies", in: [Belew and Booker 1991], pp. 2–9.

38. Bailey, J., and D. Hammerstrom (1988), "Why VLSI Implementations of Associative VLCNs Require Connection Multiplexing", in: [IEEE 1988], Vol. II, pp. 173–180.
39. Baldi, P., and Y. Chauvin (1994), "Smooth On-Line Learning Algorithms for Hidden Markov Models", *Neural Computation*, Vol. 6, No. 2, pp. 307–318.
40. Baluja, S. (1995), "An Empirical Comparison of Seven Iterative and Evolutionary Function Optimization Heuristics", Technical Report CMU-CS-95-193, Carnegie Mellon University.
41. Bandemer, H., and S. Gottwald (1989), *Einführung in Fuzzy-Methoden*, Akademie-Verlag, Berlin.
42. Barnsley, M. (1988), *Fractals Everywhere*, Academic Press, London.
43. Battiti, R. (1992), "First- and Second-Order Methods for Learning: Between Steepest Descent and Newton's Method", *Neural Computation*, Vol. 4, pp. 141–166.
44. Baum, E., and D. Haussler (1989), "What Size Network Gives Valid Generalization", *Neural Computation*, Vol. 1, pp. 151–160.
45. Baum, E. (1990a), "On Learning a Union of Half Spaces", *Journal of Complexity*, Vol. 6, pp. 67–101.
46. Baum, E. (1990b), "The Perceptron Algorithm is Fast for Nonmalicious Distributions", *Neural Computation*, Vol. 2, pp. 248–260.
47. Baum, L. E. (1972), "An Inequality and Associated Maximization Technique in Statistical Estimation for Probabilistic Functions of Markov Processes", *Inequalities III*, Academic Press, New York, pp. 1–8.
48. Becker, S., and Y. le Cun (1989), "Improving the Convergence of Back-Propagation Learning with Second Order Methods", in: [Touretzky et al. 1989], pp. 29–37.
49. Beerhold, J., M. Jansen, and R. Eckmiller (1990), "Pulse-Processing Neural Net Hardware with Selectable Topology and Adaptive Weights and Delays", in: [IEEE 1990], Vol. II, pp. 569–574.
50. Belew, R., and L. Booker (eds.) (1991), *Proceedings of the Fourth International Conference on Genetic Algorithms*, Morgan Kaufmann, San Mateo, CA.
51. Belew, R., J. McInerney, and N. Schraudolph (1992), "Evolving Networks: Using the Genetic Algorithm with Connectionist Learning", in: [Langton et al. 1992], pp. 511–547.
52. Bennett, C. (1973), "Logical Reversibility of Computation", *IBM Journal of Research and Development*, Vol. 17, No. 6, pp. 525–532.
53. Bergerson, K., and D. Wunsch (1991), "A Commodity Trading Model Based on a Neural Network-Expert System Hybrid", in: [IEEE 1991], Vol. I, pp. 289–293.
54. Bessière, P., A. Chams, A. Guerin, J. Herault, C. Jutten, and J. Lawson (1991), "From Hardware to Software: Designing a Neurostation", in: [Ramacher, Rückert 1991], pp. 311–335.
55. Bezdek, J., and S. Pal. (1992), "Fuzzy Models for Pattern Recognition – Background, Significance and Key Points", in: Bezdek, J. and Pal. S. (eds.), *Fuzzy Models for Pattern Recognition*, IEEE Press, New Jersey, pp. 1–28.
56. Bischof, H., and W. G. Kropatsch (1993), "Neural Networks Versus Image Pyramids", in: [Albrecht et al. 1993], pp. 145–153.
57. Bischof, H. (1994), *Pyramidal Neural Networks*, PhD Thesis, Technical Report IfA-TR–93–2, Technical University of Vienna.

58. Bishop, C. (1992), "Exact Calculation of the Hessian Matrix for the Multilayer Perceptron", *Neural Computation*, Vol. 4, pp. 494–501.
59. Blashfield, R., M. Aldenderfer, and L. Morey (1982), "Cluster Analysis Software", in: P. Krishnaiah, and L. Kanal (eds.), *Handbook of Statistics 2*, North-Holland, Amsterdam, pp. 245–266.
60. Block, H. (1962), "The Perceptron: A Model for Brain Functioning", *Reviews of Modern Physics*, Vol. 34, pp. 123–135. Reprinted in: [Anderson and Rosenfeld 1988].
61. Blum, A., and R. Rivest (1988), "Training a 3-Node Neural Network is NP-Complete", *Proceedings of the 1988 Annual Workshop on Computational Learning Theory*, pp. 9–18.
62. Bolc, L., and P. Borowik (1992), *Many-Valued Logics I*, Springer-Verlag, Berlin.
63. Boltjanskij, V., and V. Efremovic (1986), *Anschauliche kombinatorische Topologie*, Deutscher Verlag der Wissenschaften, Berlin.
64. Borgwardt, K. H. (1987), *The Simplex Method – A Probability Analysis*, Springer-Verlag, Berlin.
65. Bourlard, H., and N. Morgan (1993), *Connectionist Speech Recognition.* Kluwer, Boston, MA.
66. Bouton, C., M. Cottrell, J. Fort, and G. Pagés (1992), "Self-Organization and Convergence of the Kohonen Algorithm", in: *Mathematical Foundations of Artificial Neural Networks*, Sion, September 14–19.
67. Boyer, C. (1968), *A History of Mathematics*, Princeton University Press, Princeton.
68. Boynton, R. (1979), *Human Color Vision*, Holt, Rinehart and Winston, New York.
69. Braun, H. (1994), "ENZO-M: A Hybrid Approach for Optimizing Neural Networks by Evolution and Learning", in: Y. Davidor et al. (eds.) (1994), *Parallel Problem Solving from Nature*, Springer-Verlag, Berlin, pp. 440–451.
70. Brockwell, R., and R. Davis (1991), *Time Series: Theory and Methods*, Springer-Verlag, New York.
71. Bromley, K., S. Y. Kung, and E. Swartzlander (eds.) (1988), *Proceedings of the International Conference on Systolic Arrays*, Computer Society Press, Washington.
72. Brown, A. (1991), *Nerve Cells and Nervous Systems*, Springer-Verlag, Berlin.
73. Brown, T., and S. Chattarji (1995), "Hebbian Synaptic Plasticity", in: [Arbib 1995], pp. 454–459.
74. Bruck, J. (1990), "On the Convergence Properties of the Hopfield Model", *Proceedings of the IEEE*, Vol. 78, No. 10, pp. 1579–1585.
75. Bruck, J., and J. Goodman (1990), "On the Power of Neural Networks for Solving Hard Problems", *Journal of Complexity*, Vol. 6, pp. 129–135.
76. Bryson, A. E., and Y. C. Ho (1969), *Applied Optimal Control*, Blaisdell, Waltham, MA.
77. Buhmann, J. (1995), "Data Clustering and Learning", in: [Arbib 1995], pp. 278–282.
78. Burnod, Y. (1990), *An Adaptive Neural Network: The Cerebral Cortex*, Prentice-Hall, London.
79. Byrne, J., and W. Berry (eds.) (1989), *Neural Models of Plasticity*, Academic Press, San Diego, CA.

80. Cantoni, V., and M. Ferreti (1994), *Pyramidal Architectures for Computer Vision*, Plenum Press, New York.
81. Carpenter, G., and S. Grossberg (1987), "ART 2: Self-Organization of Stable Category Recognition Codes for Analog Input Patterns", *Applied Optics*, Vol. 26, pp. 4919–4930.
82. Carpenter, G., and S. Grossberg (1990), "ART 3: Hierarchical Search Using Chemical Transmitters in Self-Organizing Pattern Recognition Architectures", *Neural Networks*, Vol. 3, pp. 129–152.
83. Carpenter, G., and S. Grossberg (1991), *Pattern Recognition by Self-Organizing Neural Networks*, MIT Press, Cambridge, MA.
84. Chalmers, D. J. (1990), "The Evolution of Learning: An Experiment in Genetic Connectionism", *Proceedings of the 1990 Connectionist Models Summer School*, Morgan Kaufmann, San Mateo, CA.
85. Charniak, E. (1991), "Bayesian Networks Without Tears", *AI Magazine*, Vol. 12, No. 4, pp. 50–63.
86. Chatfield, C. (1991), *The Analysis of Time Series*, Chapman and Hall, London.
87. Connor, J., and L. Atlas (1991), "Recurrent Neural Networks and Time Series Prediction", in: [IEEE 1991], Vol. I, pp. 301–306.
88. Cook, S. (1971), "The Complexity of Theorem Proving Procedures", *Proceedings of the ACM Symposium on Theory of Computing*, ACM, New York, pp. 151–158.
89. Cooper, L. (1973), "A Possible Organization of Animal Memory and Learning", in: Lundquist, B., and S. Lundquist (eds.), *Proceedings of the Nobel Symposium on Collective Properties of Physical Systems*, New York, Academic Press, pp. 252–264.
90. Cotter, N., and T. Guillerm (1992), "The CMAC and a Theorem of Kolmogorov", *Neural Networks*, Vol. 5, No. 2, pp. 221–228.
91. Cottrell, M., and J. Fort (1986), "A Stochastic Model of Retinotopy: A Self-Organizing Process", *Biological Cybernetics*, Vol. 53, pp. 405–411.
92. Courant, R., K. Friedrichs, and H. Lewy (1928), "Über die partiellen Differenzengleichungen der mathematischen Physik", *Mathematische Annalen*, Vol. 100, pp. 32–74.
93. Courant, R. (1943), "Variational Methods for the Solution of Problems of Equilibrium and Vibrations", *Bulletin of the American Mathematical Society*, Vol. 49, No. 1, pp. 1–23.
94. Cowan, J. (1995), "Fault Tolerance", in: [Arbib 1995], pp. 390–395.
95. Crick, F. (1994), *The Astonishing Hypothesis – The Scientific Search for the Soul*, Charles Scribner's Sons, New York.
96. Croarken, M. (1990), *Early Scientific Computing in Britain*, Clarendon Press, Oxford.
97. Cronin, J. (1987), *Mathematical Aspects of Hodgkin-Huxley Theory*, Cambridge University Press, Cambridge, UK.
98. Crowder, R. (1991), "Predicting the Mackey-Glass Time Series with Cascade Correlation Learning", in: [Touretzky 1991], pp. 117-123.
99. Darius, F., and R. Rojas (1994), "Simulated Molecular Evolution or Computer Generated Artifacts?", *Biophysical Journal*, Vol. 67, pp. 2120–2122.
100. DARPA (1988), *DARPA Neural Network Study*, AFCEA International Press, Fairfax, VA.

101. David, I., R. Ginosar, and M. Yoeli (1992), "An Efficient Implementation of Boolean Functions as Self-Timed Circuits", *IEEE Transactions on Computers*, Vol. 41, No. 1, pp. 2–11.
102. Davis, L. (1987), *Genetic Algorithms and Simulated Annealing*, Morgan Kaufmann, Los Altos, CA.
103. Davis, L. (1991), "Bit-Climbing, Representational Bias, and Test Suite Design", in: [Belew and Booker 1991], pp. 18–23.
104. Dawkins, R. (1989), *The Selfish Gene*, Oxford University Press, Oxford, UK.
105. De Jong, K. (1993), "Genetic Algorithms are NOT Function Optimizers", in: Whitley (ed.), *Foundations of Genetic Algorithms 2*, Morgan Kaufmann, San Mateo, CA, pp. 5–17.
106. Deller, J., J. Proakis, and J. Hansen (1993), *Discrete Time Processing of Speech Signals*, Macmillan, Toronto.
107. Denker, J. (ed.) (1986), *Neural Networks for Computing*, AIP Conference Proceeding Series, No. 151, American Institute of Physics.
108. DeStefano, J. (1990), "Logistic Regression and the Boltzmann Machine", in: [IEEE 1990], Vol. III, pp. 199–204.
109. Deutsch, D. (1985), "Quantum Theory, the Church-Turing Principle and the Universal Quantum Computer", *Proceedings of the Royal Society of London A*, Vol. 400, pp. 97–117.
110. Devroye, L., and T. Wagner (1982), "Nearest Neighbor Methods in Discrimination", in: P. Krishnaiah, and L. Kanal (eds.), *Handbook of Statistics 2*, North-Holland, Amsterdam, pp. 193–197.
111. Dowling, J. (1987), *The Retina: An Approachable Part of the Brain*, Harvard University Press, Cambridge, MA.
112. Doyle, P., and J. Snell (1984), *Random Walks and Electric Networks*, The Mathematical Association of America, Washington.
113. Drago, G. P., and S. Ridella (1992), "Statistically Controlled Activation Weight Initialization (SCAWI)", *IEEE Transactions on Neural Networks*, Vol. 3, No. 4, pp. 627–631.
114. Dreyfus, H., and S. Dreyfus (1988), "Making a Mind Versus Modeling the Brain: Artificial Intelligence Back at a Branchpoint", in: [Graubard 1988], pp. 15–44.
115. Dreyfus, S. (1962), "The Numerical Solution of Variational Problems", *Journal of Mathematical Analysis and Applications*, Vol. 5, No. 1, pp. 30–45.
116. Dreyfus, S. E. (1990), "Artificial Neural Networks, Backpropagation and the Kelley-Bryson Gradient Procedure", *Journal of Guidance, Control and Dynamics*, Vol. 13, No. 5, pp. 926–928.
117. Durbin, R., and D. Willshaw (1987), "An Analogue Approach to the Travelling Salesman Problem Using an Elastic Net Method", *Nature*, Vol. 326, pp. 689–691.
118. Durbin, R., C. Miall, and G. Mitchison (eds.) (1989), *The Computing Neuron*, Addison-Wesley, Wokingham, UK.
119. Eberhart, R., and R. Dobbins (eds.) (1990), *Neural Network PC Tools*, Academic Press, San Diego, CA.
120. Eckmiller, R., and C. von der Malsburg (eds.) (1988), *Neural Computers*, Springer-Verlag, Berlin.
121. Eckmiller, R., G. Hartmann, and G. Hauske (eds.) (1990), *Parallel Processing in Neural Systems and Computers*, North Holland, Amsterdam.

122. Eckmiller, R., N. Goerke, and J. Hakala (1991), "Neural Networks for Internal Representation of Movements in Primates and Robots", in: [Mammone, Zeevi 1991], pp. 97–112.
123. Eckmiller, R. (1994), "Biology Inspired Pulse-Processing Neural Nets with Adaptive Weights and Delays – Concept Sources from Neuroscience vs. Applications in Industry and Medicine", in: [Zurada et al. 1994], pp. 276–284.
124. Edelman, G. (1987), *Neural Darwinism: The Theory of Neuronal Group Selection*, Basic Books, New York.
125. Efron, B. (1979), "Bootstrap Methods: Another Look at the Jackknife", *The Annals of Statistics*, Vol. 7, pp. 1–26.
126. Efron, B., and G. Gong (1983), "A Leisurely Look at the Bootstrap, the Jackknife, and Cross-Validation", *The American Statistician*, Vol. 37, No. 1, pp. 36–48.
127. Efron, B., and R. Tibshirani (1993), *An Introduction to the Bootstrap*, Chapman & Hall, New York.
128. Eigen, M. (1992), *Stufen zum Leben: Die frühe Evolution im Visier der Molekularbiologie*, Piper, Munich.
129. Evesham, H. (1986), "Origins and Development of Nomography", *Annals of the History of Computing*, Vol. 8, No. 4, pp. 324–333.
130. Fahlman, S. (1989), "Faster Learning Variations on Back-Propagation: An Empirical Study", in: [Touretzky et al. 1989], pp. 38–51.
131. Fahlman, S., and C. Lebiere (1990), "The Cascade Correlation Learning Architecture", Technical Report CMU-CS-90-100, Carnegie Mellon University.
132. Farhat, N., D. Psaltis, A. Prata, and E. Paek (1985), "Optical Implementation of the Hopfield Model", *Applied Optics*, Vol. 24, pp. 1469–1475.
133. Fausett, L. (1994), *Fundamentals of Neural Networks – Architectures, Algorithms and Applications*, Prentice Hall, Englewood Cliffs, NJ.
134. Feitelson, D. (1988), *Optical Computing: A Survey for Computer Scientists*, MIT Press, Cambridge, MA.
135. Feldman, J., and D. Ballard (1982), "Connectionist Models and Their Properties", *Cognitive Science*, Vol. 6, No. 3, pp. 205–254.
136. Feldman, J., M. Fanty, N. Goddard, and K. Lyne (1988), "Computing With Structured Connectionist Networks", *Communications of the ACM*, Vol. 31, No. 2, pp. 170–187.
137. Feldman, J., L. Cooper, C. Koch, R. Lippman, D. Rumelhart, D. Sabbah, and D. Waltz (1990), "Connectionist Systems", in: J. Traub (ed.), *Annual Review of Computer Science*, Vol. 4, pp. 369–381.
138. Feynman, R. (1963), *The Feynman Lectures on Physics*, Addison-Wesley, Reading, MA.
139. Fogel, L. J., A. J. Owens, and M. J. Walsh (1966), *Artificial Intelligence Through Simulated Evolution*, John Wiley & Sons, New York.
140. Fontanari, J. F., and R. Meir (1991), "Evolving a Learning Algorithm for the Binary Perceptron", *Network – Computation in Neural Systems*, Vol. 2, No. 4, pp. 353–359.
141. Forrest, S., and M. Mitchell (1993), "What Makes a Problem Hard for a Genetic Algorithm? Some Anomalous Results and their Explanation", *Machine Learning*, Vol. 13, No. 2–3, pp. 285–319.
142. Forrest, S., and M. Mitchell (1993), "Relative Building-Block Fitness and the Building Block Hypothesis", in: [Whitley 1993], pp. 109–126.

143. Fredkin, E., and T. Toffoli (1982), “Conservative Logic”, *International Journal of Theoretical Physics*, Vol. 21, No. 3–4, pp. 219–253.
144. Fukushima, K. (1975), “Cognitron: A Self-Organizing Multilayered Neural Network Model”, *Biological Cybernetics*, Vol. 20, pp. 121–136.
145. Fukushima, K., S. Miyake, and T. Ito (1983), “Neocognitron: A Neural Network Model for a Mechanism of Visual Pattern Recognition”, *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 13, pp. 826–834.
146. Fukushima, K., N. Wake (1991), “Handwritten Alphanumeric Character Recognition by the Neocognitron”, *IEEE Transactions on Neural Networks*, Vol. 2, No. 3, pp. 355–365.
147. Fukushima, N., M. Okada, and K. Hiroshige (1994), “Neocognitron with Dual C-Cell Layers”, *Neural Networks*, Vol. 7, No. 1, pp. 41–47.
148. Furst, M., J. B. Saxe, and M. Sipser (1981), “Parity, Circuits and the Polynomial-Time Hierarchy”, *22nd Annual Symposium on Foundations of Computer Science*, IEEE Society Press, Los Angeles, CA, pp. 260–270.
149. Gaines, B. (1977), “Foundations of Fuzzy Reasoning”, in: [Gupta, Saridis, Gaines 1977].
150. Gallant, A., and H. White (1988), “There Exists a Neural Network That Does Not Make Avoidable Mistakes”, in: [IEEE 1988], Vol. I, pp. 657–664.
151. Gallant, S. (1988), “Connectionist Expert Systems”, *Communications of the ACM*, Vol. 31, No. 2, pp. 152–169.
152. Gallant, S. I. (1990), “Perceptron-Based Learning Algorithms”, *IEEE Transactions on Neural Networks*, Vol. 1, No. 2, pp. 179–191.
153. Gallinari, P. (1995), “Training of Modular Neural Net Systems”, in: [Arbib 1995], pp. 582–585.
154. Gandy, R. (1988), “The Confluence of Ideas in 1936”, in: [Herken 1988], pp. 55–111.
155. Gardner, E. (1987), “Maximum Storage Capacity in Neural Networks”, *Europhysics Letters*, Vol. 4, pp. 481–485.
156. Garey, M., and D. Johnson (1979), *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, New York.
157. Gass, S. (1969), *Linear Programming*, McGraw-Hill, New York.
158. Geman, S., and D. Geman (1984), “Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images”, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 6, pp. 721–741.
159. Gersho, A., and R. Gray (1992), *Vector Quantization and Signal Compression*, Kluwer, Boston, MA.
160. Gibbons, A., and W. Rytter (1988), *Efficient Parallel Algorithms*, Cambridge University Press, Cambridge, UK.
161. Glickstein, M. (1988), “The Discovery of the Visual Cortex”, *Scientific American*, Vol. 259, No. 3, pp. 84–91.
162. Glover, D. (1987), “Solving a Complex Keyboard Configuration Problem Through Generalized Adaptive Search”, in: [Davis 1987], pp. 12–31.
163. Goldberg, D. (1989), *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, MA.
164. Gorges-Schleuter, M. (1989), “ASPARAGOS: An Asynchronous Parallel Genetic Optimization Strategy”, *Proceedings of the Second International Conference on Genetic Algorithms*, pp. 422–427.
165. Graubard, S. (1988), *The Artificial Intelligence Debate*, MIT Press, Cambridge, MA.

166. Grefenstette, J. (1993), "Deception Considered Harmful", in: [Whitley 1993], pp. 75–92.
167. Griffith, J. (1971), *Mathematical Neurobiology: An Introduction to the Mathematics of the Nervous System*, Academic Press, London.
168. Grossberg, S. (1976), "Adaptive Pattern Classification and Universal Pattern Recoding: I. Parallel Development and Coding of Neural Feature Detectors", *Biological Cybernetics*, Vol. 23, pp. 121–134.
169. Grossberg, S. (1988), "Nonlinear Neural Networks: Principles, Mechanisms, and Architectures", *Neural Networks*, Vol. 1, pp. 17–61.
170. Gupta, M., G. Saridis, and B. Gaines (1977), *Fuzzy Automata and Decision Processes*, North-Holland, New York.
171. Gupta, M. (1977), "Fuzzy-ism, the First Decade", in: [Gupta, Saridis, Gaines 1977].
172. Hall, L., and A. Kandel (1986), *Designing Fuzzy Expert Systems*, Verlag TÜV Rheinland, Cologne.
173. Hameroff, S. R. (1987), *Ultimate Computing – Biomolecular Consciousness and Nanotechnology*, North-Holland, Amsterdam.
174. Hameroff, S. R., J. E. Dayhoff, R. Lahoz-Beltra, A. V. Samsonovich, and S. Rasmussen (1992), "Conformational Automata in the Cytoskeleton", *Computer*, Vol. 25, No. 11, pp. 30–39.
175. Hammerstrom, D. (1990), "A VLSI Architecture for High-Performance, Low-Cost, On-Chip Learning", in: [IEEE 1990], Vol. II, pp. 537–544.
176. Hammerstrom, D., and N. Nguyen (1991), "An Implementation of Kohonen's Self-Organizing Map on the Adaptive Solutions Neurocomputer", in: [Kohonen et al. 1991], pp. 715–720.
177. Hamming, R. (1987), *Information und Codierung*, VCH, Weinheim.
178. Haken, H. (1988), *Information and Self-Organization*, Springer-Verlag, Berlin.
179. Haken, H. (1991), *Synergetic Computers and Cognition*, Springer-Verlag, Berlin.
180. Hartley, H. O. (1961), "The Modified Gauss-Newton Method for the Fitting of Non-Linear Regression Functions by Least Squares", *Technometrics*, Vol. 3, pp. 269–280.
181. Hays, W. (1988), *Statistics*, Holt, Rinehart and Winston, Fort Worth, TX.
182. Hebb, D. (1949), *The Organization of Behavior*, John Wiley, New York.
183. Hecht-Nielsen, R. (1987a), "Counterpropagation Networks", *Applied Optics*, Vol. 26, pp. 4979–4984.
184. Hecht-Nielsen, R. (1987b), "Kolmogorov's Mapping Neural Network Existence Theorem", in: [IEEE 1987], Vol. II, pp. 11–14.
185. Hecht-Nielsen, R. (1990), *Neurocomputing*, Addison-Wesley, Reading, MA.
186. Hecht-Nielsen, R. (1992), "The Munificence of High-Dimensionality", in: [Aleksander, Taylor 1992], pp. 1017–1030.
187. Hennessy, J., and D. Patterson (1990), *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Mateo, CA.
188. Herken, R. (1988), *The Universal Turing Machine: A Half Century Survey*, Kammerer & Unverzagt, Hamburg.
189. Hertz, J., A. Krogh, and R. Palmer (1991), *Introduction to the Theory of Neural Computation*, Addison-Wesley, Redwood City, CA.
190. Hille, B. (1984), *Ionic Channels of Excitable Membranes*, Sinauer Associates Inc., Sunderland, UK.

191. Hinton, G., T. Sejnowski, and D. Ackley (1984), "Boltzmann-Machines: Constrained Satisfaction Networks that Learn", CMU-CS-84-119, Carnegie Mellon University.
192. Hodges, A. (1983), *Alan Turing: The Enigma of Intelligence*, Counterpoint, London.
193. Hoekstra, J. (1991), "(Junction) Charge-Coupled Device Technology for Artificial Neural Networks", in: [Ramacher, Rückert 1991], pp. 19–46.
194. Hoffmann, N. (1991), *Simulation neuronaler Netze*, Vieweg, Braunschweig.
195. Holland, J. (1975), *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Systems*, The University of Michigan Press, Ann Arbor, MI.
196. Holmes, J. (1991), *Sprachsynthese und Spracherkennung*, Oldenbourg, Munich.
197. Holt, J., and T. Baker (1991), "Back Propagation Simulations Using Limited Precision Calculations", in: [IEEE 1991], Vol. II, pp. 121–126.
198. Hopfield, J. (1982), "Neural Networks and Physical Systems with Emergent Collective Computational Abilities", *Proceedings of the National Academy of Sciences*, Vol. 79, pp. 2554–2558.
199. Hopfield, J. (1984), "Neurons with Graded Response Have Collective Computational Properties Like those of Two-State Neurons", *Proceedings of the National Academy of Sciences*, Vol. 81, pp. 3088–3092.
200. Hopfield, J., and D. Tank (1985), "Neural Computations of Decisions in Optimization Problems", *Biological Cybernetics*, Vol. 52, pp. 141–152.
201. Hopfield, J., and D. Tank (1986), "Computing with Neural Circuits", *Science*, Vol. 233, pp. 625–633.
202. Horgan, J. (1994), "Can Science Explain Consciousness?", *Scientific American*, Vol. 271, No. 1, pp. 88–94.
203. Hornik, K., M. Stinchcombe, and H. White (1989), "Multilayer Feedforward Networks are Universal Approximators", *Neural Networks*, Vol. 2, pp. 359–366.
204. Hornik, K. (1991), "Approximation Capabilities of Multilayer Perceptrons", *Neural Networks*, Vol. 4, pp. 251–257.
205. Hubel, D. (1989), *Auge und Gehirn: Neurobiologie des Sehens*, Spektrum der Wissenschaft, Heidelberg.
206. Hush, D., J. Salas, and B. Horne (1991), "Error Surfaces for Multilayer Perceptrons", in: [IEEE 1991], Vol. I, pp. 759–764.
207. Hwang, K., and F. Briggs (1985), *Computer Architecture and Parallel Processing*, McGraw-Hill, New York.
208. Hyman, S., T. Vogl, K. Blackwell, G. Barbour, J. Irvine, and D. Alkon (1991), "Classification of Japanese Kanji Using Principal Component Analysis as a Preprocessor to an Artificial Neural Network", in: [IEEE 1991], Vol. I, pp. 233–238.
209. IEEE (1987), *IEEE International Conference on Neural Networks*, San Diego, CA, June.
210. IEEE (1988), *IEEE International Conference on Neural Networks*, San Diego, CA, July.
211. IEEE (1990), *IEEE International Joint Conference on Neural Networks*, San Diego, CA, June.
212. IEEE (1991), *IEEE International Joint Conference on Neural Networks*, Seattle, WA, July.

213. Irie, B., and S. Miyake (1988), "Capabilities of Three-Layered Perceptrons", in: [IEEE 1988], Vol. I, pp. 641–648.
214. Ising, E. (1925), "Beitrag zur Theorie des Ferromagnetismus", *Zeitschrift für Physik*, Vol. 31, No. 253.
215. Iwata, A., Y. Nagasaka, S. Kuroyagani, and N. Suzumura (1991), "Realtime ECG Data Compression Using Dual Three Layered Neural Networks for a Digital Holter Monitor", *Proceedings of the 1991 International Conference on Artificial Neural Networks*, Finland, pp. 1673–1676.
216. Jabri, M., and B. Flower (1991), "Weight Perturbation: an Optimal Architecture and Learning Technique for Analog VLSI Feedforward and Recurrent Multilayer Networks", *Neural Computation*, Vol. 3, No. 4, pp. 546–565
217. Jacobs, R. A. (1988), "Increased Rates of Convergence through Learning Rate Adaptation", *Neural Networks* Vol. 1, pp. 295–307.
218. Jacobs, R., M. Jordan, S. Nowlan, and G. Hinton (1991), "Adaptive Mixtures of Local Experts", *Neural Computation*, Vol. 3, pp. 79–87.
219. Jagota, A. (1995), "An Exercises Supplement to the Introduction to the Theory of Neural Computation", University of Memphis, FTP document.
220. Jain, A. (1989), *Fundamentals of Digital Image Processing*, Prentice-Hall, London.
221. Janikow, C., and Z. Michalewicz (1991), "An Experimental Comparison of Binary and Floating Point Representations in Genetic Algorithms", in: [Belew and Booker 1991], pp. 31–36.
222. Jetschke, G. (1989), *Mathematik der Selbstorganisation*, Vieweg, Braunschweig.
223. Ji, C., and D. Psaltis (1991), "The Capacity of Two Layer Network with Binary Weights", in: [IEEE 1991], Vol. II, pp. 127–132.
224. Johnson, D. (1987), "More Approaches to the Traveling Salesman Guide", *Nature*, Vol. 330.
225. Jolliffe, I. (1986), *Principal Component Analysis*, Springer-Verlag, New York.
226. Jones, S., K. Sammut, Ch. Nielsen, and J. Staunstrup (1991), "Toroidal Neural Network: Architecture and Processor Granularity Issues", in: [Ramacher, Rückert 1991], pp. 229–254.
227. Jordan, M., and R. Jacobs (1994), "Hierarchical Mixtures of Experts and the EM Algorithm", *Neural Computation*, Vol. 6, pp. 181–214.
228. Judd, J. S. (1988), "On the Complexity of Loading Shallow Neural Networks", *Journal of Complexity*, Vol. 4, pp. 177–192.
229. Judd, J. S. (1990), *Neural Network Design and the Complexity of Learning*, MIT Press, Cambridge, MA.
230. Judd, J. S. (1992), "Why are Neural Networks so Wide", in: [Aleksander, Taylor 1992], Vol. 1, pp. 45–52.
231. Kamp, Y., and M. Hasler (1990), *Recursive Neural Networks for Associative Memory*, John Wiley, New York.
232. Kandel, A. (1986), *Fuzzy Mathematical Techniques with Applications*, Addison-Wesley, Reading, MA.
233. Kanerva, P. (1988), *Sparse Distributed Memory*, MIT Press, Cambridge, MA.
234. Kanerva, P. (1992), "Associative-Memory Models of the Cerebellum", in: [Aleksander, Taylor 1992], pp. 23–34.
235. Karayiannis, N., and A. Venetsanopoulos (1993), *Artificial Neural Networks – Learning Algorithms, Performance Evaluation, and Applications*, Kluwer, Boston, MA.

236. Karmarkar, N. (1984), "A New Polynomial Time Algorithm for Linear Programming", *Combinatorica*, Vol. 4, No. 4, pp. 373–381.
237. Karp, R. (1972), "Reducibility Among Combinatorial Problems", in: R. Miller, and J. Thatcher (eds.), *Complexity of Computer Computations*, Plenum Press, New York, pp. 85–104.
238. Karpinski, M., and T. Werther (1993), "VC Dimension and Uniform Learnability of Sparse Polynomials and Rational Functions", *SIAM Journal on Computing*, Vol. 22, No. 6, pp. 1276–1285.
239. Kaufmann, A. (1977), "Progress in Modeling of Human Reasoning by Fuzzy Logic", in: [Gupta, Saridis, Gaines 1977], pp. 11–17.
240. Kaufmann, A., and M. Gupta (1988), *Fuzzy Mathematical Models in Engineering and Management Science*, North-Holland, Amsterdam.
241. Kelley, H. J. (1960), "Gradient Theory of Optimal Flight Paths", *ARS Journal*, Vol. 30, No. 10, pp. 947–954.
242. Keyes, R. (1982), "Communication in Computation", *International Journal of Theoretical Physics*, Vol. 21, No. 3–4, pp. 263–273.
243. Keynes, R. (1988), "Ionenkanäle in Nervenmembranen", in: [Gehirn und Nervensystem 1988], pp. 14–19.
244. Khachiyan, L. G. (1979), "A Polynomial Algorithm in Linear Programming", translated in: *Soviet Mathematics Doklady*, Vol. 20, pp. 191–194.
245. Kimoto, T., K. Asakawa, M. Yoda, and M. Takeoka (1990), "Stock-Market Prediction System with Modular Neural Networks", in: [IEEE 1990], Vol. I, pp. 1–6.
246. Kirkpatrick, S., C. Gelatt, and M. Vecchi (1983), "Optimization by Simulated Annealing", *Science*, Vol. 220, pp. 671–680.
247. Klee, V., and G. L. Minty (1972), "How Good is the Simplex Algorithm", in: O. Shisha (ed.), *Inequalities III*, Academic Press, New York, pp. 159–179.
248. Klir, G., and T. Folger (1988), *Fuzzy Sets, Uncertainty and Information*, Prentice-Hall, Englewood Cliffs, NJ.
249. Klopf, A. (1989), "Classical Conditioning Phenomena Predicted by a Drive-Reinforcement Model of Neuronal Function", in: [Byrne, Berry 1989], pp. 104–132.
250. Koch, C., J. Marroquin, and A. Yuille (1986), "Analog 'Neuronal' Networks in Early Vision", *Proceedings of the National Academy of Sciences*, Vol. 83, pp. 4263–4267.
251. Koch, C., and I. Segev (eds.) (1989), *Methods in Neuronal Modeling: From Synapses to Networks*, MIT Press, Cambridge, MA.
252. Köhle, M. (1990), *Neurale Netze*, Springer-Verlag, Vienna.
253. Kohonen, T. (1972), "Correlation Matrix Memories", *IEEE Transactions on Computers*, Vol. C-21, pp. 353–359.
254. Kohonen, T. (1982), "Self-Organized Formation of Topologically Correct Feature Maps", *Biological Cybernetics*, Vol. 43, pp. 59–69.
255. Kohonen, T. (1984), *Self-Organization and Associative Memory*, Springer-Verlag, Berlin.
256. Kohonen, T., K. Mäkisara, O. Simula, and J. Kangas (eds.) (1991), *Artificial Neural Networks*, North-Holland, Amsterdam.
257. Kojima, M., N. Megiddo, T. Noma, and A. Yoshise (1991), *A Unified Approach to Interior Point Algorithms for Linear Complementarity Problems*, Springer-Verlag, Berlin.

258. Kosko, B. (1988), “Bidirectional Associative Memories”, *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 18, pp. 49–60.
259. Kosko, B. (1992), *Neural Networks and Fuzzy Systems: A Dynamical Systems Approach to Machine Intelligence*, Prentice-Hall, London.
260. Koza, J., and J. Rice (1991), “Genetic Generation of Both the Weights and Architecture for a Neural Network”, in: [IEEE 1991], Vol. II, pp. 397–404.
261. Kramer, A., and A. Sangiovanni-Vincentelli (1989), “Efficient Parallel Learning Algorithms for Neural Networks”, in: R. Lippmann, J. Moody, and D. Touretzky (eds.) (1989), *Advances in Neural Information Processing Systems*, Vol. 1, Morgan Kaufmann, pp. 40–48.
262. Kuffler, S., and J. Nicholls (1976), *From Neuron to Brain: A Cellular Approach to the Function of the Nervous System*, Sinauer, Sunderland, UK.
263. Kung, H. T. (1988), “Systolic Communication”, in: [Bromley, Kung, Swartzlander 1988], pp. 695–703.
264. Kung, S. Y., and J. N. Hwang (1988), “Parallel Architectures for Artificial Neural Nets”, in: [IEEE 1988], Vol. II, pp. 165–172.
265. Lamb, G. (1980), *Elements of Soliton Theory*, John Wiley, New York.
266. Langton, C., C. Taylor, J. Farmer, and S. Rasmussen (eds.) (1992), *Artificial Life II*, Addison-Wesley, Redwood City, CA.
267. Lassen, N., D. Ingvar, and E. Skinhoj (1988), “Hirnfunktion und Hirndurchblutung”, in: [Gehirn und Nervensystem 1988], pp. 135–143.
268. Lavington, S. (1982), “The Manchester Mark 1”, in: [Sewiorek et al. 1982], pp. 107–109.
269. Le Cun, Y. (1985), “Une Procédure d’Apprentissage pour Réseau à Seuil Asymétrique”, in: *Cognitiva 85: A la Frontiere de l’Intelligence Artificielle des Sciences de la Conaissance des Neurosciences*, Paris, pp. 599–604.
270. Legendi, T., and T. Szentivanyi (eds.) (1983), *Leben und Werk von John von Neumann*, Bibliographisches Institut, Mannheim.
271. Lewis, P. M., and C. L. Coates (1967), *Threshold Logic*, John Wiley & Sons, New York, 1967.
272. Lim, M., and Y. Takefuji (1990), “Implementing Fuzzy Rule-Based Systems on Silicon Chips”, *IEEE Expert*, Vol. 5, No. 1, pp. 31–45.
273. Lin, C.-T., and C. Lee (1991), “Neural-Network-Based Fuzzy Logic Control and Decision System”, *IEEE Transactions on Computers*, Vol. 40, No. 12, pp. 1320–1336.
274. Lin, S., and B. Kernighan (1973), “An Effective Heuristic Algorithm for the Traveling Salesman Problem”, *Operations Research*, Vol. 21, pp. 498–516.
275. Lin, W.-M., V. Prasanna, and W. Przytula (1991), “Algorithmic Mapping of Neural Network Models onto Parallel SIMD Machines”, *IEEE Transactions on Computers*, Vol. 40, No. 12, pp. 1390–1401.
276. Linsker, R. (1988), “Self-Organization in a Perceptual Network”, *Computer*, March, pp. 105–117.
277. Lisboa, P. G., and S. J. Perantonis (1991), “Complete Solution of the Local Minima in the XOR Problem”, *Network – Computation in Neural Systems*, Vol. 2, No. 1, pp. 119–124.
278. Liu, S., J. Wu, and C. Li (1990), “Programmable Optical Threshold Logic Implementation with an Optoelectronic Circuit”, *Optics Letters*, Vol. 15, No. 12, pp. 691–693.
279. Lorentz, G. (1976), “The 13-th Problem of Hilbert”, *Proceedings of Symposia in Pure Mathematics*, Vol. 28, pp. 419–430.

280. Lovell, D. R. (1994), *The Neocognitron as a System for Handwritten Character Recognition: Limitations and Improvements*, PhD Thesis, Department of Electrical and Computer Engineering, University of Queensland.
281. Maas, H. L. van der, P. F. M. J. Verschure, and P. C. M. Molenaar (1990), "A Note on Chaotic Behavior in Simple Neural Networks", *Neural Networks*, Vol. 3, pp. 119–122.
282. MacQueen, J. (1967), "Some Methods for Classification and Analysis of Multi-Variate Observations", in: *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, University of California Press, Berkeley, CA, pp. 281–297.
283. Mahowald, M., and C. Mead (1991), "The Silicon Retina", *Scientific American*, Vol. 264, No. 5, pp. 40–47.
284. Malsburg, C. von der (1973), "Self-Organization of Orientation Sensitive Cells in the Striate Cortex", *Kybernetik*, Vol. 14, pp. 85–100.
285. Malsburg, C. von der (1986), "Frank Rosenblatt: Principles of Neurodynamics, Perceptrons and the Theory of Brain Mechanisms", in: [Palm, Aertsen 1986].
286. Mamdani, E. (1977), "Applications of Fuzzy Set Theory to Control Systems: A Survey", in: [Gupta, Saridis, Gaines 1977], pp. 77–88.
287. Mammone, R. J., and Y. Zeevi (eds.) (1991), *Neural Networks: Theory and Applications*, Academic Press, Boston, MA.
288. Mammone, R. J. (1994) (ed.), *Artificial Neural Networks for Speech and Vision*, Chapman & Hall, London.
289. Mani, D., and L. Shastri (1994), "Massively Parallel Real-Time Reasoning with Very Large Knowledge Bases – An Interim Report", International Computer Science Institute, Technical Report, TR-94-031, Berkeley, CA.
290. Mansfield, A. (1991), "Comparison of Perceptron Training by Linear Programming and by the Perceptron Convergence Procedure", in: [IEEE 1991], Vol. II, pp. 25–30.
291. Margarita, S. (1991), "Neural Networks, Genetic Algorithms and Stock Trading", [Kohonen et al. 1991], pp. 1763–1766.
292. Marquardt, D. W. (1963), "An Algorithm for the Least-Squares Estimation of Nonlinear Parameters", *Journal of the Society for Industrial and Applied Mathematics*, Vol. 11, No. 2, pp. 431–441.
293. Marr, D. (1982), *Vision – A Computational Investigation into the Human Representation and Processing of Visual Information*, W. H. Freeman, San Francisco, CA.
294. Martini, H. (1990), *Grundlagen der Assoziativen Speicherung*, BI Wissenschaftsverlag, Mannheim.
295. Matthews, G. G. (1991), *Cellular Physiology of Nerve and Muscle*, Blackwell Scientific Publications, Boston, MA.
296. McAulay, A. (1991), *Optical Computer Architectures: The Application of Optical Concepts to Next Generation Computers*, John Wiley, New York.
297. McCorduck, P. (1979), *Machines Who Think: A Personal Inquiry into the History and Prospects of Artificial Intelligence*, W. H. Freeman, New York.
298. McCulloch, W., and W. Pitts (1943), "A Logical Calculus of the Ideas Immanent in Nervous Activity", *Bulletin of Mathematical Biophysics*, Vol. 5, pp. 115–133.

299. McCulloch, W., and W. Pitts (1947), "How We Know Universals: the Perception of Auditory and Visual Forms", *Bulletin of Mathematical Biophysics*, Vol. 9, pp. 127–147.
300. McCulloch, W. (1960), "The Reliability of Biological Systems", reprinted in: *Collected Works of Warren S. McCulloch* (1989), Intersystems Publications, Salinas, Vol. 4, pp. 1193–1210.
301. McCulloch, W. (1974), "Recollections of the Many Sources of Cybernetics", reprinted in: *Collected Works of Warren S. McCulloch* (1989), Intersystems Publications, Salinas, Vol. 1, pp. 21–49.
302. Mead, C., and L. Conway (1980), *Introduction to VLSI Systems*, Addison-Wesley, Reading, MA.
303. Mead, C. (1989), *Analog VLSI and Neural Systems*, Addison-Wesley, Reading, MA.
304. Metropolis, N., A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller (1953), "Equation of State Calculations for Fast Computing Machines", *Journal of Chemical Physics*, Vol. 21, pp. 1087–1092.
305. Michalewicz, Z. (1996), *Genetic Algorithms + Data Structures = Evolution Programs*, 3rd Edition, Springer-Verlag, Berlin.
306. Mikhailov, A. (1990), *Foundations of Synergetics I: Distributed Active Systems*, Springer-Verlag, Berlin.
307. Mikhailov, A., and A. Loskutov (1991), *Foundations of Synergetics II: Complex Patterns*, Springer-Verlag, Berlin.
308. Milner, P. (1993), "The Mind and Donald O. Hebb", *Scientific American*, Vol. 268, No. 1, pp. 124–129.
309. Minsky, M. (1954), *Neural Nets and the Brain: Model Problem*, Dissertation, Princeton University, Princeton.
310. Minsky, M. (1956), "Some Universal Elements for Finite Automata", in: [Shannon and McCarthy 1956], pp. 117–128.
311. Minsky, M. (1967), *Computation: Finite and Infinite Machines*, Prentice-Hall, Englewood Cliffs, NJ.
312. Minsky, M., and S. Papert (1969), *Perceptrons: An Introduction to Computational Geometry*, MIT Press, Cambridge, MA.
313. Minsky, M. (1985), *The Society of Mind*, Simon and Schuster, New York.
314. Mitchell, M., S. Forrest, and J. H. Holland (1992), "The Royal Road for Genetic Algorithms: Fitness Landscapes and GA Performance", in: F. J. Varela, and P. Bourgine (eds.),*Toward a Practice of Autonomous Systems. Proceedings of the First European Conference on Artificial Life*, MIT Press, Cambridge, MA, pp. 245–254.
315. Møller, M. (1993), *Efficient Training of Feed-Forward Neural Networks*, PhD Thesis, Aarhus University, Denmark.
316. Montague, P. R. (1993), "The NO Hypothesis", in: B. Smith, and G. Adelman (eds.), *Neuroscience Year – Supplement 3 to the Encyclopedia of Neuroscience*, Birkhäuser, Boston, MA, pp. 100–102.
317. Montana, D., and L. Davis (1989), "Training Feedforward Neural Networks Using Genetic Algorithms", *Proceedings of the Eleventh IJCAI*, Morgan Kaufmann, San Mateo, CA, pp. 762–767.
318. Moody, J., and C. Darken (1989), "Learning with Localized Receptive Fields", in: [Touretzky et al. 1989], pp. 133–143.

319. Moody, J. (1994), "Prediction Risk and Architecture Selection for Neural Networks", in: V. Cherkassky, J. H. Friedman, and H. Wechsler (eds.), *From Statistics to Neural Networks: Theory and Pattern Recognition Applications*, NATO ASI Series F, Vol. 136, Springer-Verlag, Berlin.
320. Morell, P., and W. Norton (1988), "Myelin", in: [Gehirn und Nervensystem 1988], pp. 64–74.
321. Morgan, N., J. Beck, E. Allman, and J. Beer (1990), "RAP: A Ring Array Processor for Multilayer Perceptron Applications", *Proceedings IEEE International Conference on Acoustics, Speech, & Signal Processing*, Albuquerque, pp. 1005–1008.
322. Müller, B., J. Reinhardt, and T. M. Strickland (1995), *Neural Networks: An Introduction*, 2nd Edition, Springer-Verlag, Berlin.
323. Myers, R. (1990), *Classical and Modern Regression with Applications*, PWS-Kent Publishing Company, Boston, MA.
324. Natarajan, B. (1991), *Machine Learning – A Theoretical Approach*, Morgan Kaufmann, San Mateo, CA.
325. Neher, E., and B. Sakmann (1992), "The Patch Clamp Technique", *Scientific American*, Vol. 266, No. 3, pp. 28–35.
326. Neumann, J. von (1956), "Probabilistic Logic and the Synthesis of Reliable Organisms From Unreliable Components", in: [Shannon and McCarthy 1956], pp. 43–98.
327. Ng, K., and B. Abramson (1990), "Uncertainty Management in Expert Systems", *IEEE Expert*, April, pp. 29–47.
328. Nguyen, D., and B. Widrow (1989), "Improving the Learning Speed of 2-Layer Neural Networks by Choosing Initial Values of Adaptive Weights", *IJCNN*, pp. III–21–26.
329. Nilsson, N. (1965), *Learning Machines*, Morgan Kaufmann, San Mateo, CA, New Edition, 1990.
330. Odom, M., and R. Sharda (1990), "A Neural Network for Bankruptcy Prediction", in: [IEEE 1990], Vol. II, pp. 163–168.
331. Oja, E. (1982), "A Simplified Neuron Model as a Principal Component Analyzer", *Journal of Mathematical Biology*, Vol. 15, pp. 267–273.
332. Oja, E. (1989), "Neural Networks, Principal Components, and Subspaces", *International Journal of Neural Systems*, Vol. 1, pp. 61–68.
333. Palm, G. (1980), "On Associative Memory", *Biological Cybernetics*, Vol. 36, pp. 19–31.
334. Palm, G., and A. Aertsen (eds.) (1986), *Brain Theory*, Springer-Verlag, Berlin.
335. Parberry, I. (1994), *Circuit Complexity and Neural Networks*, MIT Press, Cambridge, MA.
336. Park, S., and K. Miller (1988), "Random Number Generators: Good Ones are Hard to Find", *Communications of the ACM*, Vol. 31, No. 10, pp. 1192–1201.
337. Parker, D. (1985), "Learning Logic", Technical Report TR–47, Center for Computational Research in Economics and Management Science, MIT, Cambridge, MA.
338. Penrose, R. (1989), *The Emperor's New Mind: Concerning Computers, Minds and the Laws of Physics*, Oxford University Press, Oxford.
339. Perrone, M. P., and L. N. Cooper (1994), "When Networks Disagree: Ensemble Methods for Hybrid Neural Networks", in: [Mammone 1994], pp. 126–142.

340. Pfister, M., and R. Rojas (1993), "Speeding-up Backpropagation – A Comparison of Orthogonal Techniques", *International Joint Conference on Neural Networks*, Nagoya, Japan, pp. 517–523.
341. Pfister, M. (1995), *Hybrid Learning Algorithms for Neural Networks*, PhD Thesis, Free University Berlin.
342. Pineda, F. (1987), "Generalization of Backpropagation to Recurrent Neural Networks", *Physical Review Letters*, Vol. 18, pp. 2229–2232.
343. Platt, J. C., and A. Barr (1987), "Constrained Differential Optimization", in: D. Anderson (ed.), *Neural Information Processing Systems 1987*, American Institute of Physics, New York, pp. 612–621.
344. Plaut, D., S. Nowlan, and G. Hinton (1986), "Experiments on Learning by Back Propagation", Technical Report CMU-CS–86–126, Carnegie Mellon University, Pittsburgh, PA.
345. Plaut, D., and T. Shallice (1994), "Word Reading in Damaged Connectionist Networks: Computational and Neuropsychological Implications", in: [Mammone 1994], pp. 294–323.
346. Pollard, W. (1986), *Bayesian Statistics for Evaluation Research*, Sage Publications, London.
347. Pomerleau, D., G. Gusciora, D. Touretzky, and H. T. Kung (1988), "Neural Network Simulation at Warp Speed: How We Got 17 Million Connections Per Second", in: [IEEE 1988], Vol. II, pp. 143–150.
348. Posner, M. (1978), *Chronometric Explorations of Mind*, Lawrence Erlbaum, Hillsdale, NJ.
349. Poston, T., C. Lee, Y. Choie, and Y. Kwon (1991), "Local Minima and Back Propagation", in: [IEEE 1991], Vol. II, pp. 173–176.
350. Poundstone, W. (1993), *Prisoner's Dilemma*, Oxford University Press, Oxford.
351. Prechelt, L. (1994), "A Study of Experimental Evaluations of Neural Network Learning Algorithms: Current Research Practice", Technical Report 19/94, University of Karlsruhe.
352. Psaltis, D., K. Wagner, and D. Brady (1987), "Learning in Optical Neural Computers", in: [IEEE 1987], Vol. III, pp. 549–555.
353. Rabiner, L., and J. Bing-Hwang (1993), *Fundamentals of Speech Recognition*, Prentice-Hall International, London.
354. Ramacher, U. (1991), "Guidelines to VLSI Design of Neural Nets", in: [Ramacher, Rückert 1991], pp. 1–17.
355. Ramacher, U., and U. Rückert (1991), *VLSI Design of Neural Networks*, Kluwer, Boston, MA.
356. Ramacher, U., J. Beichter, W. Raab, J. Anlauf, N. Bruels, U. Hachmann, and M. Wesseling (1991), "Design of a 1st Generation Neurocomputer", in: [Ramacher, Rückert 1991], pp. 271–310.
357. Ramón y Cajal, S. (1990), *New Ideas on the Structure of the Nervous System in Man and Vertebrates*, MIT Press, Cambridge, MA, translation of the French edition of 1894.
358. Rechenberg, I. (1973), *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*, Frommann-Holzboog, Stuttgart.
359. Rehkaemper, G. (1986), *Nervensysteme im Tierreich: Bau, Funktion und Entwicklung*, Quelle & Meyer, Heidelberg.
360. Reichert, H. (1990), *Neurobiologie*, Georg Thieme, Stuttgart.

361. Reingold, E., J. Nievergelt, and N. Deo (1977), *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Englewood Cliffs, NJ.
362. Rescher, N. (1969), *Many-Valued Logic*, McGraw-Hill, New York.
363. Revuz, D. (1975), *Markov Chains*, North-Holland, Amsterdam.
364. Reyneri, L., and E. Filippi (1991), "An Analysis on the Performance of Silicon Implementations of Backpropagation Algorithms for Artificial Neural Networks", *IEEE Transactions on Computers*, Vol. 40, No. 12, pp. 1380–1389.
365. Richard, M. D., and R. P. Lippmann (1991), "Neural Network Classifiers Estimate *a posteriori* Probabilities", *Neural Computation*, Vol. 3, No. 4, pp. 461–483.
366. Riedmiller, M., and H. Braun (1993), "A Direct Adaptive Method for Faster Backpropagation Learning: the Rprop Algorithm", in: *IEEE International Conference on Neural Networks*, San Francisco, CA, pp. 586-591.
367. Ritter, H., and K. Schulten (1988), "Convergence Properties of Kohonen's Topology Conserving Maps", *Biological Cybernetics*, Vol. 60, pp. 59.
368. Ritter, H., T. Martinetz, and K. Schulten (1990), *Neuronale Netze: Eine Einführung in die Neuroinformatik selbstorganisierender Netzwerke*, Addison-Wesley, Bonn.
369. Robel, A. (1995), "Using Neural Models for Analyzing Time Series of Nonlinear Dynamical Systems", *Systems Analysis Modelling Simulation*, Vol. 18–19, pp. 289–292.
370. Rojas, R. (1992), Visualisierung von neuronalen Netzen, Technical Report B 91–20, Department of Mathematics, Free University Berlin.
371. Rojas, R. (1993), "Backpropagation in General Networks", *Joint Meeting of the AMS and MAA*, San Antonio, 13–16 January.
372. Rojas, R., and M. Pfister (1993), "Backpropagation Algorithms", Technical Report B 93, Department of Mathematics, Free University Berlin.
373. Rojas, R. (1993), *Theorie der neuronalen Netze*, Springer-Verlag, Berlin.
374. Rojas, R. (1993), "A Graphical Proof of the Backpropagation Learning Algorithm", in: V. Malyshkin (ed.), *Parallel Computing Technologies, PACT 93*, Obninsk, Russia.
375. Rojas, R. (1994), "Who Invented the Computer? – The Debate from the Viewpoint of Computer Architecture", in: Gautschi, W. (ed.), *Mathematics of Computation 1943–1993*, *Proceedings of Symposia on Applied Mathematics*, AMS, pp. 361–366.
376. Rojas, R. (1994), "Oscillating Iteration Paths in Neural Networks Learning", *Computers & Graphics"*, Vol. 18, No. 4, pp. 593–597.
377. Rojas, R. (1996), "A Short Proof of the Posterior Probability Property of Classifier Neural Networks", *Neural Computation*, Vol. 8, pp. 41–43.
378. Rosenblatt, F. (1958), "The Perceptron: a Probabilistic Model for Information Storage and Organization in the Brain", *Psychological Review*, Vol. 65, pp. 386–408. Reprinted in: [Anderson and Rosenfeld 1988].
379. Rosenblatt, F. (1960), Cornell Aeronautical Laboratory Report No. VG–1196-G4, February.
380. Rosser, J. (1984), "Highlights of the History of the Lambda-Calculus", *Annals of the History of Computing*, Vol. 6, No. 4, pp. 337–349.
381. Rubner, J., and K. Schulten (1990), "Development of Feature Detectors by Self-Organization", *Biological Cybernetics*, Vol. 62, pp. 193–199.
382. Rumelhart, D., and J. McClelland (1986), *Parallel Distributed Processing*, MIT Press, Cambridge, MA.

383. Rumelhart, D., G. Hinton, and R. Williams (1986), “Learning Internal Representations by Error Propagation”, in: [Rumelhart, McClelland 1986], pp. 318–362.
384. Rumelhart, D., G. Hinton, and J. McClelland (1986), “A General Framework for Parallel Distributed Processing”, in: [Rumelhart, McClelland 1986], pp. 45–76.
385. Rumelhart, D., and D. Zipser (1986), “Feature Discovery by Competitive Learning”, in: [Rumelhart, McClelland 1986], pp. 151–193.
386. Salomon, R. (1992), *Verbesserung konnektionistischer Lernverfahren, die nach der Gradientenmethode arbeiten*, PhD Thesis, Technical University of Berlin.
387. Sanger, T. (1989), “Optimal Unsupervised Learning in a Single-Layer Linear Feedforward Neural Network”, *Neural Networks*, Vol. 2, pp. 459–473.
388. Saskin, J. (1989), *Ecken, Flächen, Kanten: Die Eulersche Charakteristik*, Deutscher Verlag der Wissenschaften, Berlin.
389. Schaffer, J. D., D. Whitley, and L. J. Eshelman (1992), “Combinations of Genetic Algorithms and Neural Networks: A Survey of the State of the Art”, *International Workshop on Combinations of Genetic Algorithms and Neural Networks*, IEEE Computer Society Press, Los Alamitos, CA, pp. 1–37.
390. Scheller, F., and F. Schubert (1989), *Biosensoren*, Birkhäuser, Basel.
391. Schiffmann, W., M. Joost, and R. Werner (1993), “Comparison of Optimized Backpropagation Algorithms”, in: M. Verleysen (ed.), *European Symposium on Artificial Neural Networks*, Brussels, pp. 97–104.
392. Schöning, U. (1987), *Logik für Informatiker*, BI Wissenschaftsverlag, Mannheim.
393. Schuster, H. (1991), *Nonlinear Dynamics and Neuronal Networks*, Proceedings of the 63rd W. E. Heraeus Seminar, VCH, Weinheim.
394. Schwefel, H. P. (1965), *Kybernetische Evolution als Strategie der experimentellen Forschung in der Strömungstechnik*, Diplomarbeit, TU-Berlin.
395. Seitz, C. (1980), “System Timing”, in: [Mead and Conway 1980], pp. 218–262.
396. Sejnowski, T., and C. Rosenberg (1986), “NETtalk: a Parallel Network that Learns to Read Aloud”, The John Hopkins University Electrical Engineering and Computer Science Technical Report, JHU/EECS–86/01.
397. Sejnowski, T., and G. Tesauro (1989), “The Hebb Rule for Synaptic Plasticity: Algorithms and Implementations”, in: [Byrne, Berry 1989], pp. 94–103.
398. Sewiorek, D., G. Bell, and A. Newell (1982), *Computer Structures: Principles and Examples*, McGraw-Hill, Auckland, NZ.
399. Shannon, C., and J. McCarthy (1956), *Automata Studies*, Princeton University Press, Princeton.
400. Shastri, L., and J. Feldman (1985), “Evidential Reasoning in Semantic Networks: A Formal Theory”, *Ninth International Joint Conference on Artificial Intelligence*, August, pp. 465–474.
401. Sheng, C. L. (1969), *Threshold Logic*, Academic Press, London.
402. Sheu, B., B. Lee, and C.-F. Chang (1991), “Hardware Annealing for Fast-Retrieval of Optimal Solutions in Hopfield Neural Networks”, in: [IEEE 1991], Vol. II, pp. 327–332.
403. Silva, F., and L. Almeida (1990), “Speeding-Up Backpropagation”, in: R. Eckmiller (ed.), *Advanced Neural Computers*, North-Holland, Amsterdam, pp. 151–156.
404. Silva, F., and L. Almeida (1991), “Speeding-up Backpropagation by Data Orthonormalization”, in: [Kohonen et al. 1991], pp. 1503–1506.

405. Siu, K.-Y., and J. Bruck (1990), “Neural Computation of Arithmetic Functions”, *Proceedings of the IEEE*, Vol. 78, No. 10, pp. 1669–1675.
406. Siu, K.-Y., and J. Bruck (1991), “On the Power of Threshold Circuits with Small Weights”, *SIAM Journal of Discrete Mathematics*, Vol. 4, No. 3, pp. 423–435.
407. Siu, K.-Y., J. Bruck, T. Kailath, and T. Hofmeister (1993), “Depth Efficient Neural Networks for Division and Related Problems” *IEEE Transactions on Information Theory*, Vol. 39, No. 3, pp. 946–956.
408. Sontag, E. (1995), “Automata and Neural Networks”, in [Arbib 1995], pp. 119–123.
409. Soucek, B., and M. Soucek (1988), *Neural and Massively Parallel Computers*, John Wiley, New York.
410. Speckmann, H., G. Raddatz, and W. Rosenstiel (1994), “Improvements of Learning Results of the Selforganizing Map by Calculating Fractal Dimensions”, in: *European Symposium on Artificial Neural Networks 94 – Proceedings*, Brussels, pp. 251–255.
411. Sprecher, D. (1964), “On the Structure of Continuous Functions of Several Variables”, *Transactions of the American Mathematical Society*, Vol. 115, pp. 340–355.
412. Stein, D. (1989), *Lectures in the Sciences of Complexity*, *Proceedings of the 1988 Complex Systems Summer School*, Addison-Wesley, Redwood City, CA.
413. Stein, D. (1989), “Spin Glasses”, *Scientific American*, Vol. 261, No. 1, pp. 36–43.
414. Steinbuch, K. (1961), “Die Lernmatrix”, *Kybernetik*, Vol. 1, No. 1, pp. 36–45.
415. Steinbuch, K. (1965), *Automat und Mensch: Kybernetische Tatsachen und Hypothesen*, Springer-Verlag, Berlin.
416. Stephens, P., and A. Goldman (1991), “The Structure of Quasicrystals”, *Scientific American*, Vol. 264, No. 4, pp. 24–31.
417. Stern, N. (1980), “John von Neumann’s Influence on Electronic Digital Computing, 1944–1946”, *Annals of the History of Computing*, Vol. 2, No. 4, pp. 349–361.
418. Stevens, C. (1988), “Die Nervenzelle”, in: [Gehirn und Nervensystem 1988], pp. 2–13.
419. Stevens, L. (1973), *Explorers of the Brain*, Angus and Robertson, London.
420. Steward, O. (1989), *Principles of Cellular, Molecular, and Developmental Neuroscience*, Springer-Verlag, New York.
421. Stone, G. (1986), “An Analysis of the Delta Rule and the Learning of Statistical Associations”, in: [Rumelhart, McClelland 1986], pp. 444–459.
422. Sugeno, M. (ed.) (1985), *Industrial Applications of Fuzzy Control*, North-Holland, Amsterdam.
423. Szu, H., and R. Hartley (1987), “Fast Simulated Annealing”, *Physics Letters A*, Vol. 122, pp. 157–162.
424. Tagliarini, G., and E. Page (1989), “Learning in Systematically Designed Networks”, in: [IEEE 1989], pp. 497–502.
425. Tesauro, G. (1990), “Neurogammon: A Neural-Network Backgammon Program”, in: [IEEE 1990], Vol. III, pp. 33–39.
426. Tesauro, G. (1995), “Temporal Difference Learning and TD-Gammon”, *Communications of the ACM*, Vol. 38, No. 3, pp. 58–68.
427. Thompson, R. (1990), *Das Gehirn: Von der Nervenzelle zur Verhaltenssteuerung*, Spektrum der Wissenschaft, Heidelberg.

428. Toffoli, T. (1980), "Reversible Computing", in: J. de Bakker, and J. van Leeuwen (eds.), *Automata, Languages and Programming*, Lecture Notes in Computer Science, Vol. 85, Springer-Verlag, Berlin.
429. Toffoli, T., and N. Margolus (1989), *Cellular Automata Machines: A New Environment for Modeling*, MIT Press, Cambridge, MA.
430. Tomlinson, M., D. Walker, and M. Sivilotti (1990), "A Digital Neural Network Architecture for VLSI", in: [IEEE 1990], Vol. II, pp. 545–550.
431. Torkkola, K., J. Kangas, P. Utela, S. Kashi, M. Kokkonen, M. Kurimo, and T. Kohonen (1991), "Status Report of the Finnish Phonetic Typewriter Project", in: [Kohonen et al. 1991], pp. 771–776.
432. Touretzky, D., G. Hinton, and T. Sejnowski (eds.) (1989), *Proceedings of the 1988 Connectionist Models Summer School*, Morgan Kaufmann, San Mateo, CA.
433. Touretzky, D., J. Elman, T. Sejnowski, and G. Hinton (eds.) (1991), *Proceedings of the 1990 Connectionist Models Summer School*, Morgan Kaufmann, San Mateo, CA.
434. Tukey, J. W. (1958), "Bias and Confidence in Not Quite Large Samples", *Annals of Mathematical Statistics*, Vol. 29, p. 614 (abstract).
435. Turing, A. (1937), "On Computable Numbers, with an Application to the Entscheidungsproblem", *Proceedings of the London Mathematical Society*, Vol. 42, pp. 230–265.
436. Valiant, L. (1984), "A Theory of the Learnable", *Communications of the ACM*, Vol. 27, pp. 1134–1142.
437. Vapnik, V., and A. Chervonenkis (1971), "On the Uniform Convergence of Relative Frequencies of Events to their Probabilities", *Theory of Probability and its Applications*, Vol. 16, pp. 264–280.
438. Vlontzos, J., and S. Y. Kung (1991), "Digital Neural Network Architecture and Implementation", in: [Ramacher, Rückert 1991], pp. 205–228.
439. Walter, J., and H. Ritter (1995), "Local PSOMs and Chebyshev PSOMs Improving the Parametrized Self-Organizing Maps", in: F. Fogelman-Soulie (ed.), *International Conference on Artificial Neural Networks*, Paris.
440. Wawrzynek, K., K. Asanovic, and N. Morgan (1993), "The Design of a Neuro-Microprocessor", *IEEE Transactions on Neural Networks*, Vol. 4, No. 3, pp. 394–399.
441. Weigend, A., and N. Gershenfeld (1994) *Time Series Prediction : Forecasting the Future and Understanding the Past*, Addison-Wesley, Reading, MA.
442. Werbos, P. (1974), *Beyond Regression – New Tools for Prediction and Analysis in the Behavioral Sciences*, PhD Thesis, Harvard University.
443. Werbos, P. (1994) *The Roots of Backpropagation – From Ordered Derivatives to Neural Networks and Political Forecasting*, J. Wiley & Sons, New York.
444. Wesseling, P. (1992), *An Introduction to Multigrid Methods*, J. Wiley & Sons, Chichester, UK.
445. Wessels, L., and E. Barnard (1992), "Avoiding False Local Minima by Proper Initialization of Connections", *IEEE Transactions on Neural Networks*, Vol. 3, No. 6, pp. 899–905.
446. White, H. (1988), "Economic Prediction Using Neural Networks: the Case of IBM Daily Stock Returns", in: [IEEE 1988], Vol. II, pp. 451–458.
447. White, H. (1992), *Artificial Neural Networks – Approximation and Learning Theory*, Blackwell, Oxford.

448. Whitley, D., S. Dominic, and R. Das (1991), "Genetic Reinforcement Learning with Multilayer Neural Networks", in: [Belew and Booker 1991], pp. 562–570.
449. Whitley, D. (ed.) (1993), *Foundations of Genetic Algorithms 2*, Morgan Kaufmann, San Mateo, CA.
450. Widrow, B., and M. Hoff (1960), "Adaptive Switching Circuits", 1960 WESCON Convention Record, New York, in: [Anderson and Rosenfeld 1989].
451. Widrow, B., and M. Lehr (1990), "30 Years of Adaptive Neural Networks: Perceptron, Madaline and Backpropagation", *Proceedings of the IEEE*, Vol. 78, No. 9, pp. 1415–1442.
452. Wiener, N. (1948), *Cybernetics*, MIT Press, Cambridge, MA.
453. Wilkinson, G. (1990), "Food-Sharing in Vampire Bats", *Scientific American*, Vol. 262, No. 2, pp. 64–71.
454. Williams, R. (1986), "The Logic of Activation Functions", in: [Rumelhart, McClelland 1986], pp. 423–443.
455. Willshaw, D., O. Buneman, and H. Longuet-Higgins (1969), "Non-Holographic Associative Memory", *Nature*, Vol. 222, pp. 960–962.
456. Wilson, G., and G. Pawley (1988), "On the Stability of the Traveling Salesman Problem Algorithm of Hopfield and Tank", *Biological Cybernetics*, Vol. 58, pp. 63–70.
457. Winder, R. (1962), *Threshold Logic*, Doctoral Dissertation, Mathematics Department, Princeton University.
458. Winograd, S., and J. Cowan (1963), *Reliable Computation in the Presence of Noise*, MIT Press, Cambridge, MA.
459. Winston, R. (1991), "Nonimaging Optics", *Scientific American*, Vol. 264, No. 3, pp. 52–57.
460. Wolfram, S. (1991), *Mathematica: A System for Doing Mathematics by Computer*, Addison-Wesley, Redwood City, CA.
461. Wooters, C. (1993), *Lexical Modeling in a Speaker Independent Speech Understanding System*, PhD Thesis, UC Berkeley, TR-93-068, International Computer Science Institute.
462. Wunsch, D., T. Caudell, D. Capps, and A. Falk (1991), "An Optoelectronic Adaptive Resonance Unit", in: [IEEE 1991], Vol. I, pp. 541–549.
463. Yamaguchi, T., and W. G. Kropatsch (1990), "Distortion-Tolerance Curve of the Neocognitron with Various Structures Including Pyramid", *IEEE 10th International Conference on Pattern Recognition*, IEEE Computer Society Press, Washington D. C., pp. 918–922.
464. Yao, A. (1985), "Separating the Polynomial-Time Hierarchy by Oracles", *26th Annual Symposium on Foundations of Computer Science*, IEEE Press, Washington D. C., pp. 1–10.
465. Zadeh, L. (1988), "Fuzzy Logic", *Computer*, Vol. 21, No. 4, April, pp. 83–93.
466. Zak, M. (1989), "Terminal Attractors in Neural Networks", *Neural Networks*, Vol. 2, pp. 259–274.
467. Zaremba, T. (1990), "Case Study III: Technology in Search of a Buck", in: [Eberhart, Dobbins 1990], pp. 251–283.
468. Zhang, S., and A. G. Constantinides (1992), "Lagrange Programming Neural Networks", *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, Vol. 39, No. 7, pp. 441–52.
469. Zurada, J. (1992), *Introduction to Artificial Neural Systems*, West Publishing Company, St. Paul, MN.

470. Zurada, J. M., R. Marks II, and C. Robinson (1994) (eds.), *Computational Intelligence – Imitating Life*, IEEE Press, New York.
471. Zurek, W. (1990), *Complexity, Entropy and the Physics of Information*, Santa Fe Institute, Studies in the Sciences of Complexity, Addison-Wesley, Redwood City, CA.
472. – (1990), *Gehirn und Kognition*, Spektrum der Wissenschaft, Heidelberg.
473. – (1988), *Gehirn und Nervensystem*, Spektrum der Wissenschaft, Heidelberg.