CHAPTER 4

NEUROCOMPUTING

"Inside our heads is a magnificent structure that controls our actions and somehow evokes an awareness of the world around ... It is hard to see how an object of such unpromising appearance can achieve the miracles that we know it to be capable of."
(R. Penrose, The Emperor's New Mind, Vintage, 1990; p. 483)

"Of course, something about the tissue in the human brain is necessary for our intelligence, but the physical properties are not sufficient ... Something in the patterning of neural tissue is crucial."
(S. Pinker, How the Mind Works, The Softback Preview, 1998; p. 65)

4.1 INTRODUCTION

How does the brain process information? How is it organized? What are the biological mechanisms involved in brain functioning? These form just a sample of some of the most challenging questions in science. Brains are especially good at performing functions like pattern recognition, (motor) control, perception, flexible inference, intuition, and guessing. But brains are also slow, imprecise, make erroneous generalizations, are prejudiced, and are incapable of explaining their own actions.

Neurocomputing, sometimes called *brain-like computation* or *neurocomputation*, but most often referred to as *artificial neural networks* (ANN)[1], can be defined as information processing systems (computing devices) designed with inspiration taken from the nervous system, more specifically the brain, and with particular emphasis on problem solving. S. Haykin (1999) provides the following definition:

> "A[n artificial] neural network is a massively parallel distributed processor made up of simple processing units, which has a natural propensity for storing experiential knowledge and making it available for use." (Haykin, 1999; p. 2)

Many other definitions are available, such as

> "A[n artificial] neural network is a circuit composed of a very large number of simple processing elements that are neurally based." (Nigrin, 1993; p. 11)
>
> "… neurocomputing is the technological discipline concerned with parallel, distributed, adaptive information processing systems that develop infor-

[1] Although neurocomputing can be viewed as a field of research dedicated to the *design* of brain-like computers, this chapter uses the word neurocomputing as a synonym to artificial neural networks.

> mation processing capabilities in response to exposure to an information environment. The primary information processing structures of interest in neurocomputing are neural networks…" (Hecht-Nielsen, 1990, p. 2)

Neurocomputing systems are distinct from what is now known as *computational neuroscience*, which is mainly concerned with the development of biologically-based computational models of the nervous system. Artificial neural networks, on the other hand, take a loose inspiration from the nervous system and emphasize the problem solving capability of the systems developed. However, most books on computational neuroscience not only acknowledge the existence of artificial neural networks, but also use several ideas from them in the proposal of more biologically plausible models. They also discuss the ANN suitability as models of real biological nervous systems.

Neurons are believed to be the basic units used for computation in the brain, and their simplified abstract models are the basic processing units of neurocomputing devices or artificial neural networks. Neurons are connected to other neurons by a small junction called synapse, whose capability of being modulated is believed to be the basis for most of our cognitive abilities, such as perception, thinking, and inferring. Therefore, some essential information about neurons, synapses, and their structural anatomy are relevant for the understanding of how ANNs are designed taking inspiration from biological neural networks.

The discussion to be presented here briefly introduces the main aspects of the nervous system used to devise neurocomputing systems, and then focuses on some of the most commonly used artificial neural networks, namely, single- and multi-layer perceptrons, self-organizing networks, and Hopfield networks. The description of the many algorithms uses a matrix notation particularly suitable for the software implementation of the algorithms. Appendix B.1 provides the necessary background on linear algebra. The biological plausibility of each model is also discussed.

4.2 THE NERVOUS SYSTEM

All multicellular organisms have some kind of nervous system, whose complexity and organization varies according to the animal type. Even relatively simple organisms, such as worms, slugs, and insects, have the ability to learn and store information in their nervous systems. The nervous system is responsible for informing the organism through sensory input with regards to the environment in which it lives and moves, processing the input information, relating it to previous experience, and transforming it into appropriate actions or memories.

The nervous system plays the important role of processing the incoming information (signals) and providing appropriate actions according to these signals. The elementary processing units of the nervous system are the *neurons*, also called *nerve cells*. *Neural networks* are formed by the interconnection of many neurons. Each neuron in the human brain has on the order of hundreds or thousands of connections to other neurons.

Anatomically, the nervous system has two main divisions: *central nervous system* (CNS) and *peripheral nervous system* (PNS), the distinction being their different locations. Vertebrate animals have a *bony spine* (*vertebral column*) and a *skull* (*cranium*) in which the central parts of the nervous system are housed. The peripheral part extends throughout the remainder of the body. The part of the (central) nervous system located in the skull is referred to as the *brain*, and the one found in the spine is called the *spinal cord*. The brain and the spinal cord are continuous through an opening in the base of the skull; both are in contact with other parts of the body through the nerves.

The brain can be further subdivided into three main structures: the *brainstem*, the *cerebellum,* and the *forebrain*, as illustrated in Figure 4.1. The *brainstem* is literally the stalk of the brain through which pass all the nerve fibers relaying input and output signals between the spinal cord and higher brain centers. It also contains the cell bodies of neurons whose axons go out to the periphery to innervate the muscles and glands of the head. The structures within the brainstem are the *midbrain*, *pons,* and the *medulla*. These areas contribute to functions such as breathing, heart rate and blood pressure, vision, and hearing. The cerebellum is located behind the brainstem and is chiefly involved with skeletal muscle functions and helps to maintain posture and balance and provides smooth, directed movements. The *forebrain* is the large part of the brain remaining when the brainstem and cerebellum have been excluded. It consists of a central core, the *diencephalon*, and right and left *cerebral hemispheres* (the *cerebrum*).

The outer portion of the cerebral hemispheres is called *cerebral cortex*. The cortex is involved in several important functions such as thinking, voluntary movements, language, reasoning, and perception. The *thalamus* part of the diencephalon is important for integrating all sensory input (except smell) before it is presented to the cortex. The *hypothalamus*, which lies below the thalamus, is a tiny region responsible for the integration of many basic behavioral patterns, which involve correlation of neural and endocrine functions. Indeed, the hypothalamus appears to be the most important area to regulate the internal environment (homeostasis). It is also one of the brain areas associated with emotions.

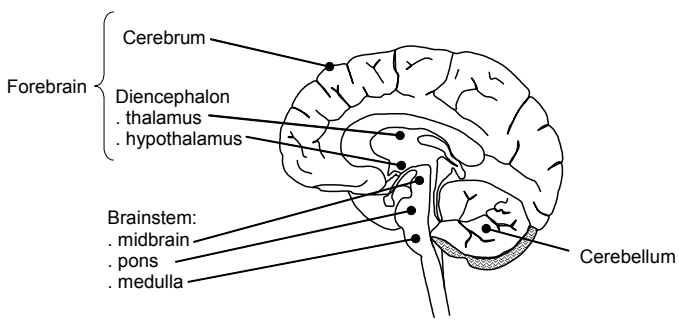


Figure 4.1: Structural divisions of the brain as seen in a midsagittal section.

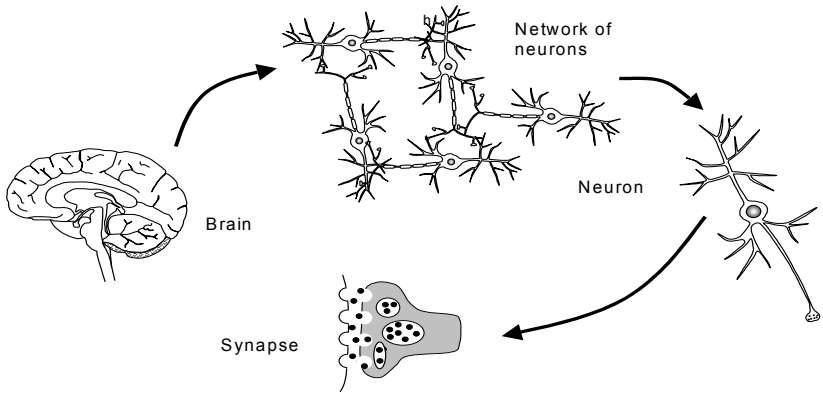


Figure 4.2: Some levels of organization in the nervous system.

4.2.1. Levels of Organization in the Nervous System

What structures really constitute a level of organization in the nervous system is an empirical not an *a priori* matter. We cannot tell, in advance of studying the nervous system, how many levels there are, nor what is the nature of the structural and functional features of any given level (Churchland and Sejnowski, 1992). Therefore, only the structures particularly interesting for the understanding, description, and implementation of artificial neural networks will be described here.

The nervous system can be organized in different levels: molecules, synapses, neurons, networks, layers, maps, and systems (Figure 4.2). An easily recognizable structure in the nervous system is the neuron, which is a cell specialized in signal processing. Depending on environmental conditions, the neurons are capable of generating a signal, more specifically an *electric potential*, that is used to transmit information to other cells to which it is connected. Some processes in the neuron utilize cascades of biochemical reactions that influence information processing in the nervous system. Many neuronal structures can be identified with specific functions. For instance, the *synapses* are important for the understanding of signal processing in the nervous system (Trappenberg, 2002).

Neurons and Synapses

Neurons use a variety of specialized biochemical mechanisms for information processing and transmission. These include *ion channels* that allow a controlled influx and outflux of currents, the generation and propagation of *action potentials*, and the release of *neurotransmitters*. Signal transmission between neurons is the core of the information processing capabilities of the brain. One of the most exciting discoveries in neuroscience was that the effectiveness of the signal

transmission can be modulated in various ways, thus allowing the brain to adapt to different situations. It is believed to be the basis of associations, memories, and many other mental abilities (Trappenberg, 2002). *Synaptic plasticity*, that is the capability of synapses to be modified, is a key ingredient in most models described in this chapter.

Figure 4.3 shows a picture of a schematic generic neuron labeling its most important structural parts. The biological neuron is a single *cell*, thus containing a *cell body* with a *nucleus* (or *soma*) containing DNA, it is filled with fluid and cellular organelles, and is surrounded by a *cell membrane*, just like any other cell in the body. Neurons also have specialized extensions called *neurites*, that can be further distinguished into *dendrites* and *axons*. While the dendrites receive signals from other neurons, the axon propagates the output signal to other neurons.

One peculiarity about neurons is that they are specialized in signal processing utilizing special electrophysical and chemical processes. They can receive and send signals to many other neurons. The neurons that send signals, usually termed *sending* or *presynaptic neurons*, contact the *receiving* or *postsynaptic neurons* in specialized sites named *synapses* either at the cell body or at the dendrites. The synapse is thus the junction between the presynaptic neuron's axon and the postsynaptic neuron's dendrite or cell body.

The general information processing feature of synapses allow them to alter the state of a postsynaptic neuron, thus eventually triggering the generation of an electric pulse, called *action potential*, in the postsynaptic neuron. The action potential is usually initiated at the *axon hillock* and travels all along the axon, which can finally branch and send information to different regions of the nervous system. Therefore, a neuron can be viewed as a device capable of receiving diverse input stimuli from many other neurons and propagating its single output response to many other neurons.

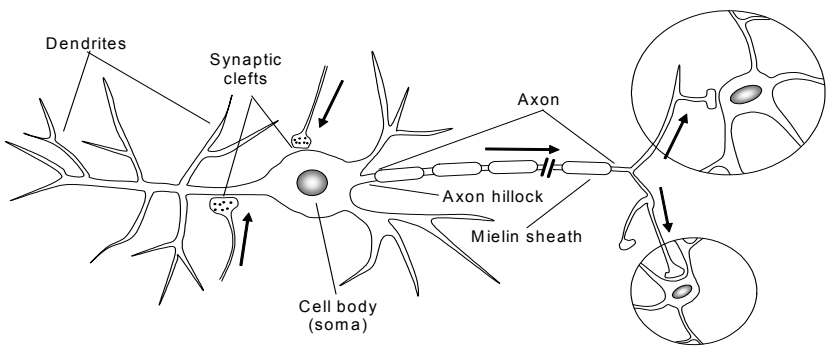


Figure 4.3: Schematic neuron similar in appearance to the pyramidal cells in the brain cortex. The parts outlined are the major structural components of most neurons. The direction of signal propagation between and within neurons is shown by the dark arrows.

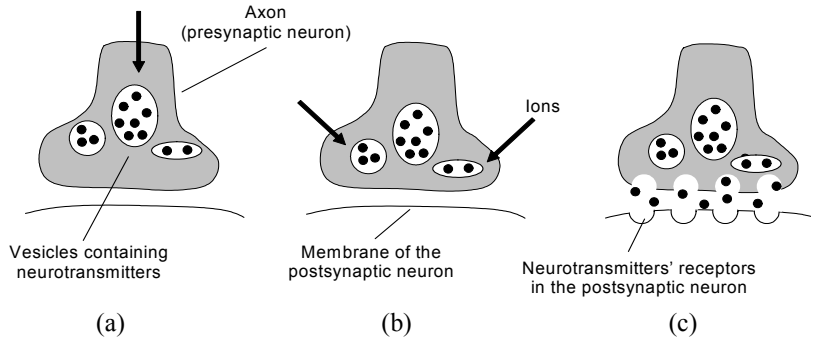


Figure 4.4: Diagram of a chemical synapse. The action potential arrives at the synapse (a) and causes ions to be mobilized in the axon terminal (b), thus causing vesicles to release neurotransmitters into the cleft, which in turn bind with the receptors on the postsynaptic neurons (c).

Various mechanisms exist to transfer information (signals) among neurons. As neurons are cells encapsulated in membranes, little openings in these membranes, called *channels*, allow the transfer of information among neurons. The basic mechanisms of information processing are based on the movement of charged atoms, *ions*, in and out of the channels and within the neuron itself. Neurons live in a liquid environment in the brain containing a certain concentration of ions, and the flow of ions in and out of a neuron through channels is controlled by several factors. A neuron is capable of altering the intrinsic electrical potential, the *membrane potential*, of other neurons, which is given by the difference between the electrical potential within and in the surroundings of the cell.

When an action potential reaches the terminal end of an axon, it mobilizes some ions by opening voltage-sensitive channels that allow the flow of ions into the terminal and possibly the release of some of these stored ions. These ions then promote the release of *neurotransmitters* (chemical substances) into the *synaptic cleft*, which finally diffuse across the cleft and binds with receptors in the postsynaptic neurons. As outcomes, several chemical processes might be initiated in the postsynaptic neuron, or ions may be allowed to flow into it. Figure 4.4 summarizes some of the mechanisms involved in synapse transmission.

As the electrical effects of these ions (action potential) propagate through the dendrites of the receiving neuron and up to the cell body, the process of information can begin again in the postsynaptic neuron. When ions propagate up to the cell body, these signals are *integrated* (summed), and the resulting membrane potential will determine if the neuron is going to *fire*, i.e., to send an output signal to a postsynaptic neuron. This only occurs if the membrane potential of the neuron is greater than the neuron *threshold*. This is because the channels are particularly sensitive to the membrane potential; they will only open when

the potential is sufficiently large (O'Reilly and Munakata, 2000). The action of neural firing is also called *spiking*, *firing a spike*, or the *triggering of an action potential*. The spiking is a very important electrical response of a neuron; once generated it does not change its shape with increasing current. This phenomenon is called the *all-or-none* aspect of the action potential.

Different types of neurotransmitters and their associated ion channels have distinct effects on the state of the postsynaptic neuron. One class of neurotransmitters opens channels that will allow positively charged ions to enter the cell, thus triggering the increase of the membrane potential that drives the postsynaptic neurons towards their excited state. Other neurotransmitters initiate processes that drive the postsynaptic potential towards a resting state; a potential known as the *resting potential*. Therefore, neurotransmitters can promote the initiation of *excitatory* or *inhibitory* processes.

Networks, Layers, and Maps

Neurons can have *forward* and *feedback* connections to other neurons, meaning that they can have either one way or reciprocal connections with other neurons in the nervous system. These interconnected neurons give rise to what is known as *networks of neurons* or *neural networks*. For instance, within a cubic millimeter of cortical tissue, there are approximately 10^5 neurons and about 10^9 synapses, with the vast majority of these synapses arising from cells located within the cortex (Churchland and Sejnowski, 1992). Therefore, the degree of interconnectivity in the nervous system is quite high.

A small number of interconnected neurons (units) can exhibit complex behaviors and information processing capabilities that cannot be observed in single neurons. One important feature of neural networks is the representation of information (knowledge) in a *distributed* way, and the *parallel processing* of this information. No single neuron is responsible for storing "a whole piece" of knowledge; it has to be distributed over several neurons (connections) in the network. Networks with specific architectures and specialized information processing capabilities are incorporated into larger structures capable of performing even more complex information-processing tasks.

Many brain areas display not only networks of neurons but also *laminar organization*. Laminae are *layers of neurons* in register with other layers, and a given lamina conforms to a highly regular pattern of where it projects to and from where it receives projections. For instance, the *superior colliculus* (a particular layered midbrain structure; see Figure 4.1) receives visual inputs in superficial layers, and in deeper layers it receives tactile and auditory input. Neurons in an intermediate layer of the superior colliculus represent information about eye movements (Churchland and Sejnowski, 1992).

One of the most common arrangements of neurons in the vertebrate nervous systems is a layered two-dimensional structure organized with a *topographic* arrangement of unit responses. Perhaps the most well-known example of this is the mammalian cerebral cortex, though others such as the superior colliculus also use maps. The cerebral cortex is the outside surface of the brain. It is a two-

dimensional structure extensively folded with fissures and hills in many larger or more intelligent animals. Two different kinds of cortex exist: an older form with three sub-layers called *paleocortex*, and a newer form that is most prominent in animals with more complex behavior, a structure with six or more sub-layers called *neocortex*.

Studies in human beings by neurosurgeons, neurologists, and neuropathologists have shown that different cortical areas have separate functions. Thus, it is possible to identify visible differences between different regions of the cerebral cortex, each presumably corresponding to a processing module. Figure 4.5 illustrates some specific functional areas in the cerebral cortex of humans. Note that regions associated with different parts of the body can have quite sharp boundaries.

In general, it is known that neocortical neurons are organized into six distinct layers, which can be subdivided into *input*, *hidden*, and *output* layer. The input layer usually receives the sensory input, the output layer sends commands and outputs to other portions of the brain, and the hidden layers receive inputs locally from other cortical layers. This means that hidden layers neither directly receive sensory stimuli nor produce motor and other outputs.

One major principle of organization within many sensory and motor systems is the *topographic map*. For instance, neurons in visual areas of the cortex (in the rear end of the cortex, opposite to the eyes; see Figure 4.5) are arranged topographically, in the sense that adjacent neurons have adjacent visual receptive fields and collectively they constitute a map of the retina. Because neighboring processing units are concerned with similar representations, topographic mapping is an important means whereby the brain manages to save on wiring and also to share wire.

Networking, topographic mapping, and layering are all special cases of a more general principle: the exploitation of geometric and structural properties in information processing design. Evolution has shaped our brains so that these structural organizations became efficient ways for biological systems to assemble in one place information needed to solve complex problems.

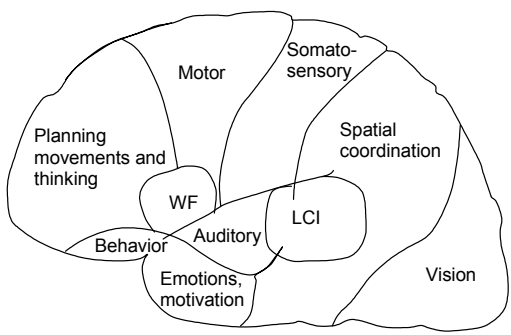


Figure 4.5: Map of specific functional areas in the cerebral cortex. WF: word formation; LCI: Language comprehension and intelligence.

4.2.2. Biological and Physical Basis of Learning and Memory

The nervous system is continuously modifying and updating itself. Virtually all its functions, including perception, motor control, thermoregulation, and reasoning, are modifiable by experience. The topography of the modifications appears not to be final and finished, but an ongoing process with a virtually interminable schedule. Behavioral observations indicate degrees of plasticity in the nervous system: there are fast and easy changes, slower and deeper modifiability, and more permanent but still modifiable changes.

In general, global learning is a function of local changes in neurons. There are many possible ways a neuron could change to embody adaptations. For instance, new dendrites might sprout out, or there might be extension of existing branches, or existing synapses could change, or new synapses might be created. In the other direction, pruning could decrease the dendrites or bits of dendrites, and thus decrease the number of synapses, or the synapses on the remaining branches could be shut down altogether. These are all postsynaptic changes in the dendrites. There could also be changes in the axons; for instance, there might be changes in the membrane, or new branches might be formed, and genes might be induced to produce new neurotransmitters or more of the old ones. Presynaptic changes could include changes in the number of vesicles released per spike and the number of transmitter molecules contained in each vesicle. Finally, the whole cell might die; taking with it all the synapses it formerly supported (Churchland and Sejnowsky, 1992).

This broad range of structural adaptability can be conveniently condensed in the present discussion by referring simply to synapses, since every modification either involves synaptic modification directly or indirectly, or can be reasonably so represented. Learning by means of setting synaptic efficiency is thus the most important mechanism in neural networks, biological and artificial. It depends on both individual neuron-level mechanisms and network-level principles to produce an overall network that behaves appropriately in a given environment.

Two of the primary mechanisms underlying learning in the nervous system are the *long-term potentiation* (LTP) and *long-term depression* (LTD), which refer to the strengthening and weakening of weights in a nontransient form. Potentiation corresponds to an increase in the measured depolarization or excitation delivered by a controlled stimulus onto a receiving neuron, and depression corresponds to a decrease in the measured depolarization. In both cases, the excitation or inhibition of a membrane potential may trigger a complex sequence of events that ultimately result in the modification of synaptic efficiency (strength).

Such as learning, *memory* is an outcome of an adaptive process in synaptic connections. It is caused by changes in the synaptic efficiency of neurons as a result of neural activity. These changes in turn cause new pathways or facilitated pathways to develop for transmission of signals through the neural circuits of the brain. The new or facilitated pathways are called *memory traces*; once established, they can be activated by the thinking mind to reproduce the memories. Actually, one of the outcomes of a learning process can be the creation of a

more permanent synaptic modification scheme, thus resulting in the memorization of an experience.

Memories can be classified in a number of ways. One common classification being into:

- *Short-term memory*: lasts from a few seconds to a few minutes, e.g., one's memory of a phone number.
- *Intermediate long-term memory*: lasts from minutes to weeks, e.g., the name of a nice girl/boy you met in a party.
- *Long-term memory*: lasts for an indefinite period of time, e.g., your home address.

While the two first types of memories do not require many changes in the synapses, long-term memory is believed to require *structural changes* in synapses. These structural changes include an increase in number of vesicle release sites for secretion of neurotransmitter substances, an increase in the number of presynaptic terminals, an increase in the number of transmitter vesicles, and changes in the structures of the dendritic spines.

Therefore, the difference between learning and memory may be sharp and conceptual. Learning can be viewed as the adaptive process that results in the change of synaptic efficiency and structure, while memory is the (long-lasting) result of this adaptive process.

4.3 ARTIFICIAL NEURAL NETWORKS

Artificial neural networks (ANN) present a number of features and performance characteristics in common with the nervous system:

- The basic information processing occurs in many simple elements called (artificial) *neurons*, *nodes* or *units*.
- These neurons can *receive* and *send stimuli* from and to other neurons and the environment.
- Neurons can be connected to other neurons thus forming networks of neurons or *neural networks*.
- Information (signals) are transmitted between neurons via connection links called *synapses*.
- The efficiency of a synapse, represented by an associated *weight value* or *strength*, corresponds to the information stored in the neuron, thus in the network.
- Knowledge is acquired from the environment by a process known as *learning*, which is basically responsible for *adapting* the connection strengths (weight values) to the environmental stimuli.

One important feature of artificial neural networks is where the knowledge is stored. Basically, what is stored is the connection strengths (synaptic strengths) between units (artificial neurons) that allow patterns to be recreated. This feature

has enormous implications, both for processing and learning. The knowledge representation is set up so that the knowledge necessarily influences the course of processing; it becomes a part of the processing itself. If the knowledge is incorporated into the strengths of the connections, then learning becomes a matter of finding the appropriate connection strengths so as to produce satisfactory patterns of activation under some circumstances.

This is an extremely important feature of ANNs, for it opens up the possibility that an information processing mechanism could *learn*, by tuning its connections strengths, to capture the interdependencies between activations presented to it in the course of processing. Another important implication of this type of representation is that *the knowledge is distributed* over the connections among a large number of units. There is no 'special' unit reserved for particular patterns.

An artificial neural network can be characterized by three main features: 1) a set of *artificial neurons*, also termed *nodes*, *units*, or simply *neurons*; 2) the pattern of connectivity among neurons, called the network *architecture* or *structure*; and 3) a method to determine the weight values, called its *training* or *learning* algorithm. Each one of these features will be discussed separately in the following sections.

4.3.1. Artificial Neurons

In the biological neuron, inputs come into the cell primarily through channels located in synapses, allowing ions to flow into and out of the neuron. A membrane potential appears as a result of the integration of the neural inputs, and will then determine whether a given neuron will produce a spike (action potential) or not. This spike causes neurotransmitters to be released at the end of the axon, which then forms synapses with the dendrites of other neurons. The action potential only occurs when the membrane potential is above a critical threshold level. Different inputs can provide different amounts of activation depending on how much neurotransmitter is released by the sender and how many channels in the postsynaptic neuron are opened. Therefore, there are important features of the biological synapses involved in the information processing of neurons.

The net effect of these biological processes is summarized in the computational models discussed here by a *weight* (also called *synaptic strength*, *synaptic efficiency*, *connection strength*, or *weight value*) between two neurons. Furthermore, the modification of one or more of these weight factors will have a major contribution for the neural network learning process. This section reviews three models of neuronal function. The first is the McCulloch-Pitts model in which the neuron is assumed to be computing a logic function. The second is a simple analog *integrate-and-fire* model. And the third is a generic connectionist neuron, which integrates its inputs and generates an output using one particular *activation function*. These neuronal models may be interchangeably called *nodes*, *units*, *artificial neurons*, or simply neurons. It is important to have in mind, though, that the nodes most commonly used in artificial neural networks bear a far resemblance with real neurons.

The McCulloch and Pitts Neuron

W. McCulloch and W. Pitts (1943) wrote a famous and influential paper based on the computations that could be performed by two-state neurons. They did one of the first attempts to understand nervous activity based upon elementary neural computing units, which were highly abstract models of the physiological properties of neurons and their connections. Five physical assumptions were made for their calculus (McCulloch and Pitts, 1943):

1. The behavior of the neuron is a binary process.
2. At any time a number of synapses must be excited in order to activate the neuron.
3. Synaptic delay is the only significant delay that affects the nervous system.
4. The excitation of a certain neuron at a given time can be inhibited by an inhibitory synapse.
5. The neural network has a static structure; that is, a structure that does not change with time.

McCulloch and Pitts conceived the neuronal response as being equivalent to a *proposition* adequate to the neuron's stimulation. Therefore, they studied the behavior of complicated neural networks using a notation of the symbolic *logic of propositions* (see Appendix B.4.2). The 'all-or-none' law of nervous activity was sufficient to ensure that the activity of any neuron could be represented as a proposition.

It is important to note that according to our current knowledge of how the neuron works - based upon electrical and chemical processes - neurons are not realizing any proposition of logic. However, the model of McCulloch and Pitts can be considered as a special case of the most general neuronal model to be discussed in the next sections, and is still sometimes used to study particular classes of nonlinear networks. Additionally, this model has caused a major impact mainly among computer scientists and engineers, encouraging the development of artificial neural networks. It has even been influential in the history of computing (cf. von Neumann, 1982).

The McCulloch and Pitts neuron is binary, i.e., it can assume only one of two states (either '0' or '1'). Each neuron has a fixed *threshold* θ and receives inputs from synapses of identical weight values. The neuronal mode of operation is simple. At each time step t, the neuron responds to its synaptic inputs, which reflect the state of the presynaptic neurons.

If no inhibitory synapse is active, the neuron integrates (sums up) its synaptic inputs, generating the *net input* to the neuron, u, and checks if this sum (u) is greater than or equal to the threshold θ. If it is, then the neuron becomes active, that is, responds with a '1' in its output ($y = 1$); otherwise it remains inactive, that is, responds with a '0' in its output ($y = 0$).

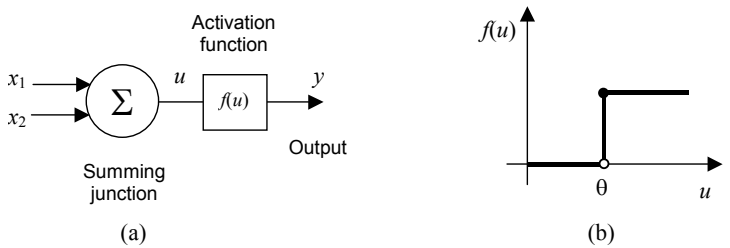


Figure 4.6: The McCulloch and Pitts neuron (a) and its threshold activation function (b).

a	b	a AND b
0	0	0
0	1	0
1	0	0
1	1	1

a	b	a OR b
0	0	0
0	1	1
1	0	1
1	1	1

a	NOT a
0	1
1	0

Figure 4.7: Truth tables for the connectives AND, OR, and NOT.

Although this neuron is quite simple, it already presents some important features in common with most neuron models, namely, the integration of the input stimuli to determine the *net input u* and the presence of an activation function (threshold). Figure 4.6 illustrates the simple McCulloch and Pitts neuron and its activation function.

To illustrate the behavior of this simple unit, assume two excitatory inputs x_1 and x_2 and a threshold $\theta = 1$. In this case, the neuron is going to fire; that is, to produce an output '1', every time x_1 or x_2 has a value '1', thus operating like the logical connective OR (see Figure 4.7). Assume now that the neuron threshold is increased to $\theta = 2$. In this new situation, the neuron is only going to be active (fire) if both x_1 and x_2 have value '1' simultaneously, thus operating like the logical connective AND (see Figure 4.7).

A Basic Integrate-and-Fire Neuron

Assume a noise-free neuron with the net input being a variable of time $u(t)$ corresponding to the membrane potential of the neuron. The main effects of some neuronal channels (in particular the sodium and leakage channels) can be captured by a simple equation of an integrator (Dayan and Abbot, 2001; Trappenberg, 2002):

$$\tau_m \frac{du(t)}{dt} = u_{res} - u(t) + R_m i(t) \qquad \textbf{(4.1)}$$

where τ_m is the membrane time constant of the neuron determined by the average conductance of the channels (among other things); u_{res} is the resting potenti-

al of the neuron; i is the input current given by the sum of the synaptic currents generated by firings of presynaptic neurons; R_m is the resistance of the neuron to the flow of current (ions); and t is the time index.

Equation (4.1) can be very simply understood. The rate of variation of the membrane potential of the neuron is proportional to its current membrane potential, its resting potential, and the potential generated by the incoming signals to the neuron. Note that the last term on the right side of this equation is the Ohm's law ($u = R.i$) for the voltage generated by the incoming currents.

The input current $i(t)$ to the neuron is given by the sum of the incoming synaptic currents depending on the *efficiency* of individual synapses, described by the variable w_j for each synapse j. Therefore, the total input current to the neuron can be written as the sum of the individual synaptic currents multiplied by a weight value

$$i(t) = \sum_j \sum_{t_j^f} w_j f(t - t_j^f) \tag{4.2}$$

where the function $f(\cdot)$ parameterizes the form of the postsynaptic response. This function was termed *activation function* in the McCulloch and Pitts neuron discussed above and this nomenclature will be kept throughout this text. The variable t_j^f denotes the firing time of the presynaptic neuron of synapse j. The firing time of the postsynaptic neuron is defined by the time the membrane potential u reaches a threshold value θ,

$$u(t^f) = \theta \tag{4.3}$$

In contrast to the firing time of the presynaptic neurons, the firing time of the integrate-and-fire neuron has no index. To complete the model, the membrane potential has to be reset to the resting state after the neuron has fired. One form of doing this is by simply resetting the membrane potential to a fixed value u_{res} immediately after a spike.

The Generic Neurocomputing Neuron

The computing element employed in most neural networks is an integrator, such as the McCulloch and Pitts and the integrate-and-fire models of a neuron, and computes based on its connection strengths. Like in the brain, the artificial neuron is an information-processing element that is fundamental to the operation of the neural network. Figure 4.8 illustrates the typical artificial neuron, depicting its most important parts: the synapses, characterized by their *weight values* connecting each input to the neuron; the *summing junction* (*integrator*); and the *activation function*.

Specifically, an input signal x_j at the input of synapse j connected to neuron k is multiplied by the synaptic weight w_{kj}. In this representation, the first subscript of the synaptic weight refers to the neuron, and the second subscript refers to the synapse connected to it. The summing junction adds all input signals weighted by the synaptic weight values plus the neuron's bias b_k; this operation constitutes the dot (or inner) product (see Appendix B.1); that is, a linear combination of the inputs with the weight values, plus the bias b_k. Finally, an activation function is

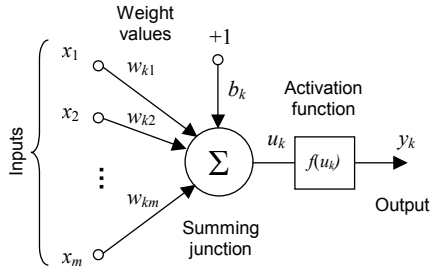


Figure 4.8: Nonlinear model of a neuron.

used to limit the amplitude of the output of the neuron. The activation function is also referred to as a *squashing function* (Rumelhart et al., 1986) for it limits the permissible amplitude range of the output signal to some finite value.

The bias has the effect of increasing or decreasing the net input to the activation function depending on whether it is positive or negative, respectively. In the generic neuron it is usually used in place of a fixed threshold θ for the activation function. For instance, the McCulloch and Pitts neuron (Figure 4.6) will fire when its net input is greater than θ:

$$y = f(u) = \begin{cases} 1 & \text{if } u \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

where $u = x_1 + x_2$ for the neuron of Figure 4.6. It is possible to replace the threshold θ by a bias weight b that will be multiplied by a constant input of value '1':

$$y = f(u) = \begin{cases} 1 & \text{if } u \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

where $u = x_1 + x_2 - b$. Note however, that while the bias is going to be adjusted during learning, the threshold assumes a fixed value.

It is important to note here that the output of this generic neuron is simply a number, and the presence of discrete action potentials is ignored. As real neurons are limited in dynamic range from zero-output firing rate to a maximum of a few hundred action potentials per second, the use of an activation function could be biologically justified.

Mathematically, the neuron k can be described by a simple equation:

$$y_k = f(u_k) = f\left(\sum_{j=1}^{m} w_{kj} x_j + b_k \right) \qquad \textbf{(4.4)}$$

where x_j, $j = 1, \ldots, m$, are the input signals; w_{kj}, $j = 1, \ldots, m$, are the synaptic weights of neuron k; u_k is the net input to the activation function; b_k is the bias of neuron k; $f(\cdot)$ is the activation function; and y_k is the output signal of the neuron.

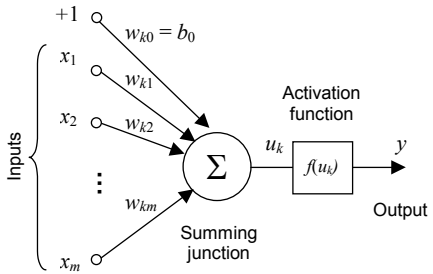


Figure 4.9: Reformulated model of a neuron.

It is possible to simplify the notation of Equation (4.4) so as to account for the presence of the bias by simply defining a constant input signal $x_0 = 1$ connected to the neuron k with associated weight value $w_{k0} = b_k$. The resulting equation becomes,

$$y_k = f(u_k) = f\left(\sum_{j=0}^{m} w_{kj} x_j\right) \tag{4.5}$$

Figure 4.9 illustrates the reformulated model of a neuron.

Types of Activation Function

The activation function, denoted by $f(u_k)$, determines the output of a neuron k in relation to its net input u_k. It can thus assume a number of forms, of which some of the most frequently used are summarized below (see Figure 4.10):

- *Linear function*: relates the net input directly with its output.

$$f(u_k) = u_k \tag{4.6}$$

- *Step or threshold function*: the output is '1' if the net input is greater than or equal to a given threshold θ; otherwise it is either '0' or '−1', depending on the use of a binary {0,1} or a bipolar {−1,1} function. The binary step function is the one originally used in the McCulloch and Pitts neuron:

$$f(u_k) = \begin{cases} 1 & \text{if } u_k \geq \theta \\ 0 & \text{otherwise} \end{cases} \tag{4.7}$$

- *Signum function*: it is similar to the step function, but has a value of '0' when the net input to the neuron is $u = 0$. It can also be either bipolar or binary; the bipolar case being represented in Equation (4.8).

$$f(u_k) = \begin{cases} 1 & \text{if } u_k > \theta \\ -1 & \text{if } u_k < \theta \end{cases} \tag{4.8}$$

- *Sigmoid function*: it is a strictly increasing function that presents saturation and a graceful balance between linear and nonlinear behavior. This is the most commonly used activation function in the ANN literature. It has an

s-shaped form and can be obtained by several functions, such as the *logistic function*, the *arctangent function*, and the *hyperbolic tangent* function; the difference being that, as u_k ranges from $-\infty$ to $+\infty$, the logistic function ranges from 0 to 1, the arctangent ranges from $-\pi/2$ to $+\pi/2$, and the hyperbolic tangent ranges from –1 to +1:

$$Logistic: \ f(u_k) = \frac{1}{1+\exp(-u_k)} \tag{4.9}$$

- *Radial basis function*: it is a nonmonotonic function that is symmetric around a base value. This is the main type of function for *radial basis function neural networks*. The expression shown below is that of the Gaussian bell curve:

$$f(u_k) = \exp(-u_k^2) \tag{4.10}$$

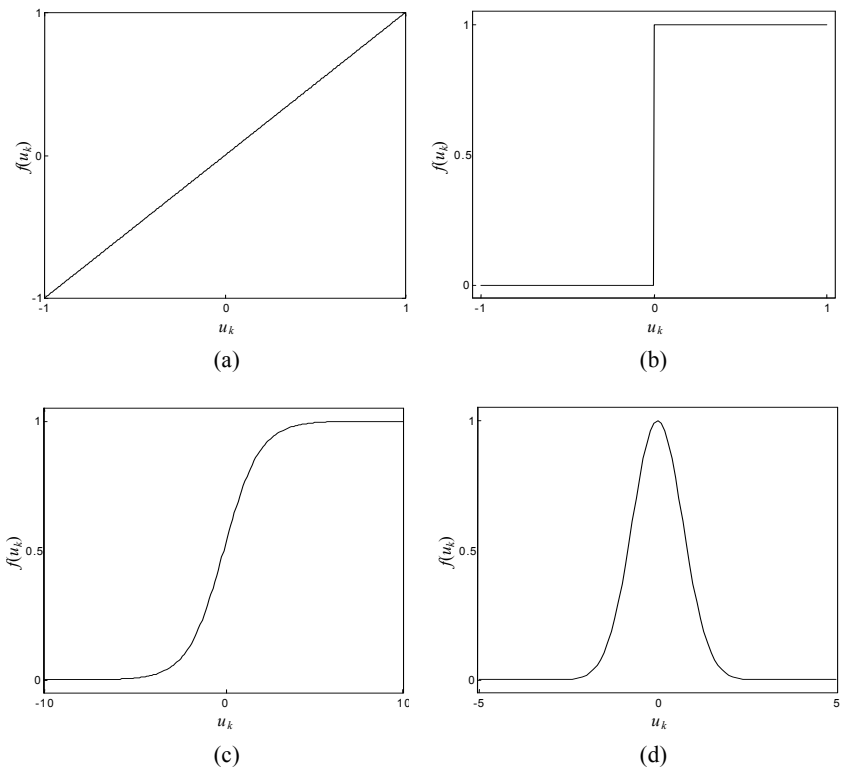


Figure 4.10: Most commonly used types of activation function. (a) Linear activation, $u_k \in [-1,+1]$. (b) Step function with $\theta = 0$, $u_k \in [-1,+1]$. (c) Logistic function, $u_k \in [-10,+10]$. (d) Gaussian bell curve, $u_k \in [-5,+5]$.

It is important to stress that the functions described and illustrated in Figure 4.10 depict only the general shape of those functions. However, it should be clear that these basic shapes can be modified. All these functions can include parameters with which some features of the functions can be changed, such as the *slope* and *offset* of a function. For instance, by multiplying u_k in the logistic function by a value β, it is possible to control the smoothness of the function:

$$f(u_k) = \frac{1}{1+\exp(-\beta.u_k)}$$

In this case, for larger values of β, this function becomes more similar to the threshold function, while for smaller values of β, the logistic function becomes more similar to a linear function.

4.3.2. Network Architectures

One of the basic prerequisites for the emergence of complex behaviors is the interaction of a number of individual agents. In the nervous system, it is known that individual neurons can affect the behavior (firing rate) of others, but, as a single entity, a neuron is meaningless. This might be one reason why evolution drove our brains to a system with such an amazingly complex network of interconnected neurons.

In the human brain not much is known of how neurons are interconnected. Some particular knowledge is available for specific brain portions, but little can be said about the brain as a whole. For instance, it is known that the cortex can be divided into a number of different cortical areas specialized in different kinds of processing; some areas perform pattern recognition, others process spatial information, language processing, and so forth. In addition, it is possible to define anatomic neuronal organizations in the cortex in terms of *layers of neurons*, which are very important for the understanding of the detailed biology of the cortex (Figure 4.5).

In the domain of artificial neural networks, a layer of neurons will refer to functional layers of nodes. As not much is known about the biological layers of neurons, most neurocomputing networks employ some standardized architectures, specially designed for the engineering purposes of solving problems. There are basically three possible layers in a network: an *input layer*, one or more *intermediate* (or *hidden*) *layers*, and an *output layer*. These are so-called because the input layer receives the input stimuli directly from the environment, the output layer places the network output(s) to the environment, and the hidden layer(s) is not in direct contact with the environment.

The way in which neurons are structured (interconnected) in an artificial neural network is intimately related with the learning algorithm that is used for training the network. In addition, this interconnectivity affects the network storage and learning capabilities. In general, it is possible to distinguish three main types of network architectures (Haykin, 1999): *single-layer feedforward networks*, *multi-layer feedforward networks,* and *recurrent networks*.

Single-Layer Feedforward Networks

The simplest case of layered networks consists of an input layer of nodes whose output feeds the output layer. Usually, the input nodes are linear, i.e., they simply propagate the input signals to the output layer of the network. In this case, the input nodes are also called sensory units because they only sense (receive information from) the environment and propagate the received information to the next layer. In contrast, the output units are usually processing elements, such as the neuron depicted in Figure 4.9, with a nonlinear type of activation function. The signal propagation in this network is purely positive or *feedforward*; that is, signals are propagated from the network input to its outputs and never the opposite way (*backward*). This architecture is illustrated in Figure 4.11(a) and the direction of signal propagation in Figure 4.11(b). In order not to overload the picture, very few connection strengths were depicted in Figure 4.11(a), but these might be sufficient to give a general idea of how the weights are assigned in the network.

The neurons presented in Figure 4.11 are of the generic neuron type illustrated in Figure 4.9. (Note that input x_0 is assumed to be fixed in '1' and the weight vector that connects it to all output units w_{i0}, $i = 1, \ldots, o$, corresponds to the bias of each output node $b_i = w_{i0}$, $i = 1, \ldots, o$.) The weight values between the inputs and the output nodes can be written using matrix notation (Appendix B.1) as

$$\mathbf{W} = \begin{bmatrix} w_{10} & w_{11} & \cdots & w_{1m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{o0} & w_{o1} & \cdots & w_{om} \end{bmatrix} \tag{4.11}$$

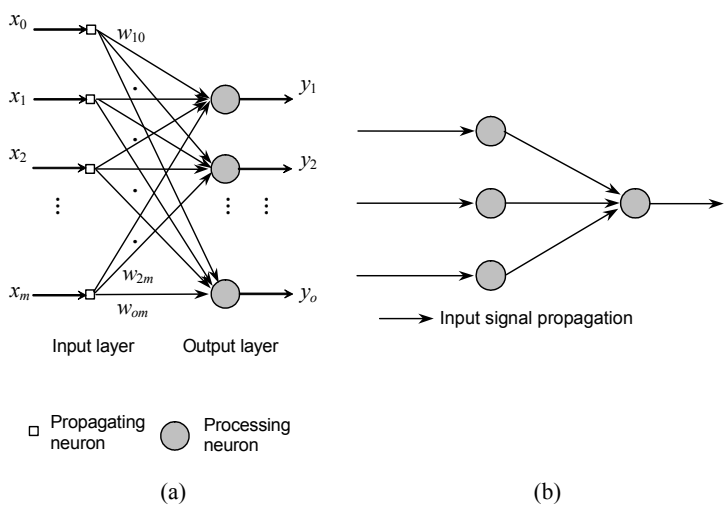


Figure 4.11: *Single-layer feedforward* neural network. (a) Network architecture. (b) Direction of propagation of the input signal.

where the first index i, $i = 1, \ldots, o$, of each element corresponds to the postsynaptic node and the second index j, $j = 0, \ldots, m$, corresponds to the presynaptic node (just remember 'to-from' while reading from left to right). Therefore, each row of matrix $\mathbf{W}$ corresponds to the weight vector of an output unit, and each element of a given column j, $j = 0, \ldots, m$, corresponds to the strength with which an input j is connected to each output unit i. Remember that $j = 0$ corresponds to the unitary input that will multiply the neuron's bias.

The output of each postsynaptic neuron i in Figure 4.11(a), $i = 1, \ldots, o$, is computed by applying an activation function $f(\cdot)$ to its net input, which is given by the linear combination of the network inputs and the weight vector,

$$y_i = f(\mathbf{w}_i.\mathbf{x}) = f(\Sigma_j \, w_{ij}.x_j) \, , j = 0, \ldots, m \qquad \textbf{(4.12)}$$

The output vector of the whole network $\mathbf{y}$ is given by the activation function applied to the product of the weight matrix $\mathbf{W}$ by the input vector $\mathbf{x}$,

$$\mathbf{y} = f(\mathbf{W}.\mathbf{x}) \qquad \textbf{(4.13)}$$

Assuming the general case where the weight and input vectors can take any real value, the dimension of the weight matrix and each vector is $\mathbf{W} \in \Re^{o \times (m+1)}$, $\mathbf{w}_i \in \Re^{1 \times (m+1)}$, $i = 1, \ldots, o$, $\mathbf{x} \in \Re^{(m+1) \times 1}$, and $\mathbf{y} \in \Re^{o \times 1}$.

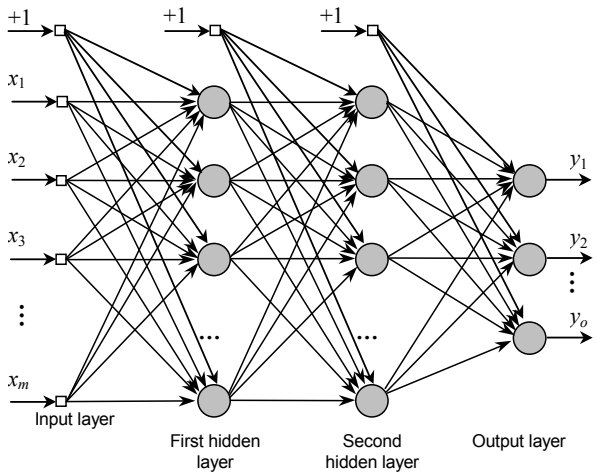
Multi-Layer Feedforward Networks

The second class of feedforward neural networks is known as multi-layer networks. These are distinguished from the single-layered networks by the presence of one or more *intermediate* or *hidden* layers. By adding one or more hidden layers, the nonlinear computational processing and storage capability of the network is increased, for reasons that will become clearer further in the text. The output of each network layer is used as input to the following layer. A multi-layer feedforward network can be thought of as an assembly line: some basic material is introduced into the production line and passed on, the second stage components are assembled, and then the third stage, up to the last or output stage, which delivers the final product.

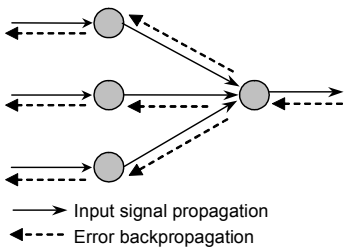
The learning algorithm typically used to train this type of network requires the *backpropagation* of an error signal calculated between the network output and a desired output. This architecture is illustrated in Figure 4.12(a) and the direction of signal propagation is depicted in Figure 4.12(b).

In such networks, there is one weight matrix for each layer, and these are going to be denoted by the letter $\mathbf{W}^k$ with a superscript k indicating the layer. Layers are counted from left to right, and the subscripts of this matrix notation remain the same as previously. Therefore, w_{ij}^k corresponds to the weight value connecting the postsynaptic neuron i to the presynaptic neuron j at layer k.

In the network of Figure 4.12(a), $\mathbf{W}^1$ indicates the matrix of connections between the input layer and the first hidden layer; matrix $\mathbf{W}^2$ contains the connections between the first hidden layer and the second hidden layer; and matrix $\mathbf{W}^3$ contains the connections between the second hidden layer and the output layer.



(a)



(b)

Figure 4.12: *Multi-layer feedforward* neural network. (a) Network architecture (legend as depicted in Figure 4.11). (b) Direction of propagation of the functional and error signals. (The weight values were suppressed from the picture not to overload it.)

In feedforward networks the signals are propagated from the inputs to the network output in a layer-by-layer form (from left to right). Therefore, the network output is given by (in matrix notation):

$$\mathbf{y} = \mathbf{f}^3(\mathbf{W}^3\ \mathbf{f}^2(\mathbf{W}^2\ \mathbf{f}^1(\mathbf{W}^1\mathbf{x}))) \qquad \textbf{(4.14)}$$

where $\mathbf{f}^k$ is the vector of activation functions of layer k (note that nodes in a given layer may have different activation functions); $\mathbf{W}^k$ is the weight matrix of layer k and $\mathbf{x}$ is the input vector.

It can be observed from Equation (4.14) that the network output is computed by recursively multiplying the weight matrix of a given layer by the output produced by each previous layer. The expression to calculate the output of each layer is given by Equation (4.12) using the appropriate weight matrix and inputs.

It is important to note that if the intermediate nodes have linear activation functions there is no point in adding more layers to this network, because $f(x) = x$ for linear functions, and thus the network output would be given by

$$\mathbf{y} = \mathbf{f}^3(\mathbf{W}^3\,\mathbf{W}^2\,\mathbf{W}^1\mathbf{x}).$$

and this expression can be reduced to,

$$\mathbf{y} = \mathbf{f}^3(\mathbf{W}\mathbf{x}),$$

where $\mathbf{W} = \mathbf{W}^3\,\mathbf{W}^2\,\mathbf{W}^1$.

Therefore, if one wants to increase the computational capabilities of a multilayer neural network by adding more layers to the network, nonlinear activation functions have to be used in the hidden layers.

Recurrent Networks

The third class of networks is known as recurrent networks, distinguished from feedforward networks for they have at least one *recurrent* (or *feedback*) loop. For instance, a recurrent network may consist of a single layer of neurons with each neuron feeding its output signal back to the input of other neurons, as illustrated in Figure 4.13. In a feedback arrangement, there is communication between neurons; there can be cross talk and plan revision; intermediate decisions can be taken; and a mutually agreeable solution can be found.

Note that the type of feedback loop illustrated in Figure 4.13 is distinguished from the backpropagation of error signals briefly mentioned in the previous section. The recurrent loop has an impact on the network learning capability and performance because it involves the use of particular branches composed of retard units (Z^{-1}) resulting in a nonlinear dynamic behavior (assuming that the network has nonlinear units).

The network illustrated in Figure 4.13 is fully recurrent, in the sense that all network units have feedback connections linking them to all other units in the network. This network also assumes an input vector **x** weighted by the weight vector **w** (not shown).

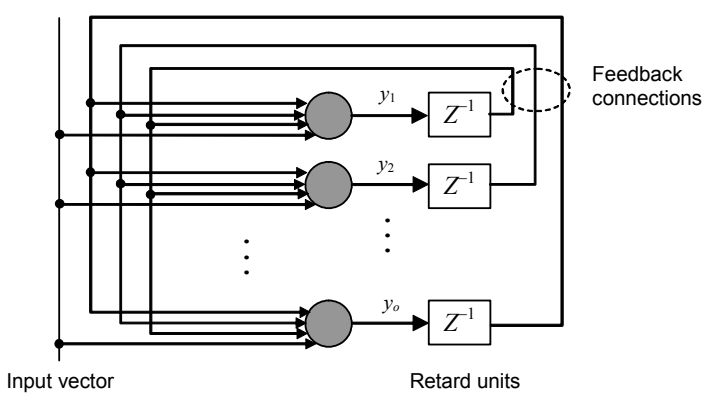


Figure 4.13: Recurrent neural network with no hidden layer.

The network output y_i, $i = 1, \ldots, o$, is a composed function of the weighted inputs at iteration t, plus the weighted outputs in the previous iteration $(t - 1)$:

$$y_i(t) = f(\mathbf{w}_i.\mathbf{x}(t) + \mathrm{v}_i.\mathbf{y}(t-1)) = f(\Sigma_j\, w_{ij}.x_j(t) + \Sigma_k\, v_{ik}.y_k(t-1)) \,, \quad j = 1, \ldots, m;\ k = 1, \ldots, o \tag{4.15}$$

where $\mathbf{v}_i$, $i = 1, \ldots, o$ is the vector weighting the retarded outputs fed back into the network ($\mathbf{W} \in \Re^{o \times m}$, $\mathbf{w}_i \in \Re^{1 \times m}$, $i = 1, \ldots, o$, $\mathbf{x} \in \Re^{m \times 1}$, $\mathbf{y} \in \Re^{o \times 1}$, $\mathbf{V} \in \Re^{o \times o}$, and $\mathbf{v}_i \in \Re^{1 \times o}$).

4.3.3. Learning Approaches

It has been discussed that one of the most exciting discoveries in neuroscience was that synaptic efficiency could be modulated to the input stimuli. Also, this is believed to be the basis for learning and memory in the brain. Learning thus involves the adaptation of synaptic strengths to environmental stimuli. Biological neural networks are known not to have their architectures much altered throughout life. New neurons cannot routinely be created to store new knowledge, and the number of neurons is roughly fixed at birth, at least in mammals. The alteration of synaptic strengths may thus be the most relevant factor for learning. This change could be the simple modification in strength of a given synapse, the formation of new synaptic connections or the elimination of pre-existing synapses. However, the way learning is accomplished in the biology of the brain is still not much clear.

In the neurocomputing context, *learning* (or *training*) corresponds to the process by which the network's "free parameters" are adapted (adjusted) through a mechanism of presentation of environmental (or input) stimuli. In the standard neural network learning algorithms to be discussed here, these free parameters correspond basically to the connection strengths (weights) of individual neurons. It is important to have in mind though, that more sophisticated learning algorithms are capable of dynamically adjusting several other parameters of an artificial neural network, such as the network architecture and the activation function of individual neurons. The environmental or input stimuli correspond to a set of *input data* (or *patterns*) that is used to *train* the network.

Neural network learning basically implies the following sequence of events:

- Presentation of the input patterns to the network.
- Adaptation of the network free parameters so as to produce an altered pattern of response for the input data.

In most neural network applications the network weights are first adjusted according to a given learning rule or algorithm, and then the network is applied to a new set of input data. In this case there are two steps involved in the use of a neural network: 1) network training, and 2) network application.

With standard learning algorithms a neural network learns through an iterative process of weight adjustment. The type of learning is defined by the way in which the weights are adjusted. The three main learning approaches are: 1) *supervised learning*, 2) *unsupervised learning*, and 3) *reinforcement learning*.

Supervised Learning

This learning strategy embodies the concept of a *supervisor* or *teacher*, who has the knowledge about the environment in which the network is operating. This knowledge is represented in the form of a set of *input-output samples* or *patterns*. Supervised learning is typically used when the class of data is known *a priori* and this can be used as the supervisory mechanism, as illustrated in Figure 4.14. The network free parameters are adjusted through the combination of the input and error signals, where the error signal is the difference between the desired output and the current network output.

To provide the intuition behind supervised learning, consider the following example. Assume you own an industry that produces patterned tiles. In the quality control (QC) part of the industry, an inspection has to be made in order to guarantee the quality in the patterns printed on the tiles. There are a number of tiles whose patterns are assumed to be of good quality, and thus pass the QC. These tiles can be used as input samples to train the artificial neural network to classify good quality tiles. The known or desired outputs are the respective classification of the tiles as good or bad quality tiles.

In the artificial neural network implementation, let neuron j be the only output unit of a feedforward network. Neuron j is stimulated by a signal vector $\mathbf{x}(t)$ produced by one or more hidden layers, that are also stimulated by an input vector. Index t is the discrete time index or, more precisely, the time interval of an iterative process that will be responsible for adjusting the weights of neuron j. The only output signal $y_j(t)$, from neuron j, is compared with a *desired output*, $d_j(t)$. An error signal $e_j(t)$ is produced:

$$e_j(t) = d_j(t) - y_j(t) \qquad \textbf{(4.16)}$$

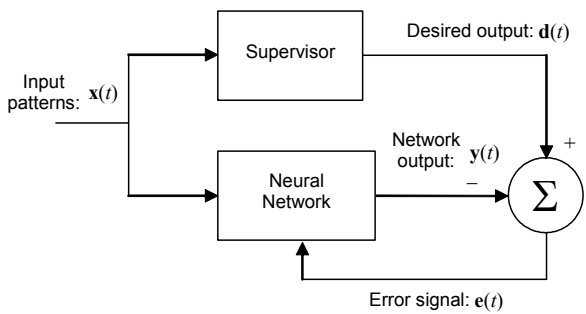


Figure 4.14: In supervised learning, the environment provides input patterns to train the network and a supervisor has the knowledge about the environment in which the network is operating. At each time step t, the network output is compared with the desired output (the error is used to adjust the network response).

The error signal acts as a control mechanism responsible for the application of corrective adjustments in the weights of neuron *j*. The goal of supervised learning is to make the network output more similar to the desired output at each time step; that is, to *correct the error* between the network output and the desired output. This objective can be achieved by minimizing a *cost function* (also called *performance index*, *error function*, or *objective function*), $\Im(t)$, which represents the instant value of the error measure:

$$\Im(t) = \frac{1}{2} e_j^2(t) \qquad \textbf{(4.17)}$$

Generalization Capability

Once the supervised learning process is complete, it is possible to present an *unseen* pattern to the network, which will then classify that pattern, with a certain degree of accuracy, into one of the classes used in the training process. This is an important aspect of an artificial neural network, its *generalization capability*. In the present context, *generalization* refers to the performance of the network on (new) patterns that were not used in the network learning process.

This can be illustrated with the help of the following example. Assume we want to use a multi-layer feedforward neural network to approximate the function $sin(x).cos(2x)$ depicted by a solid line in Figure 4.15(a). It is only necessary a few training samples to adjust the weight vectors of a multi-layer feedforward neural network with a single hidden layer composed of five sigmoidal units. The network contains a single input and a single output unit, but these details are not relevant for the present discussion.

It is known that most real-world data contains noise, i.e., irrelevant or meaningless data, or data with disturbances. To investigate the relationship between noisy data and the network capability of approximating these data, some noise was added to the input data by simply adding a uniform distribution of zero mean and variance 0.15 to each input datum. Figure 4.15(a) shows the input data and the curve representing the original noise-free function $sin(x).cos(2x)$. Note that the noisy data generated deviates a little from the function to be approximated.

In this case, if the neural network is not trained sufficiently or if the network architecture is not appropriate, the approximation it will provide for the input data will not be satisfactory; an *underfitting* will occur (Figure 4.15(b)). In the opposite case, when the network is trained until it perfectly approximates the input data, some *overfitting* will occur, meaning that the approximation is too accurate for this noisy data set (Figure 4.15(c)). A better training is the one that establishes a compromise between the approximation accuracy and a good generalization for unseen data (Figure 4.15(d)).

From a biological perspective, generalization is very important for our creation of models of the world. Think of generalization according to the following example. If you just memorize some specific facts about the world instead of trying to extract some simple essential regularity underlying these facts, then you would be in trouble when dealing with novel situations where none of the specifics appear. For instance, if you memorize a situation where you almost

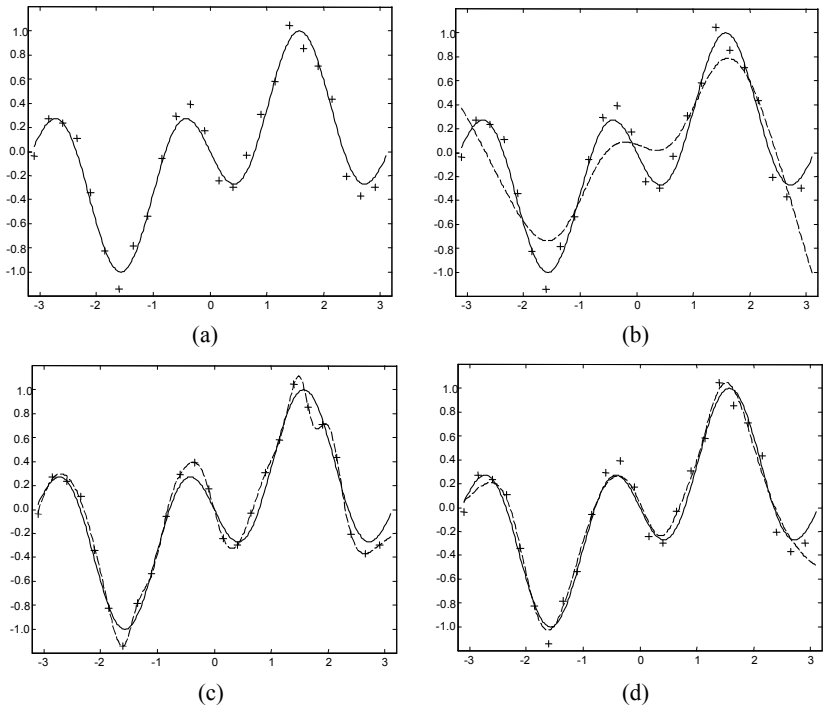


Figure 4.15: Function $sin(x) \times cos(2x)$ with uniformly distributed noise over the interval [-0.15, 0.15]. Legend: + input patterns; — desired output; - - - network output. (a) Training patterns with noise and the original function to be approximated (solid line). (b) Training process interrupted too early, *underfitting* (dashed line). (c) Too much training, *overfitting* (dashed line). (d) Better trade-off between quality of approximation and generalization capability.

drowned in seawater (but you survived) as 'stay away from seawater', then you may be in trouble if you decide to swim in a pool or a river. Of course, if the problem is that you don't know how to swim, then, knowing that seawater in particular is dangerous might not be a sufficiently good 'model' of the world to prevent you from drowning in other types of water. Your internal model has to be general enough so as to suggest 'stay away from water'.

Unsupervised Learning

In the *unsupervised* or *self-organized* learning approach, there is no supervisor to evaluate the network performance in relation to the input data set. The intuitive idea embodied in unsupervised learning is quite simple: given a set of input data, what can you do with it? For instance, if you are given a set of balloons, you could group them by colors, or by shape, or by any other attribute you can qualify.

Note that in unsupervised learning there is no error information being fed back into the network; the classes of the data are *unknown* or *unlabelled* and the presence of the supervisor no longer exists. The network adapts itself to statistical regularities in the input data, developing an ability to create internal representations that encode the features of the input data and thus, generate new classes automatically. Usually, self-organizing algorithms employ a *competitive learning* scheme.

In *competitive learning*, the network output neurons compete with one another to become activated, with a single output neuron being activated at each iteration. This property makes the algorithm appropriate to discover salient statistical features within the data that can then be used to classify a set of input patterns.

Individual neurons learn to specialize on groups (*clusters*) of similar patterns; in effect they become *feature extractors* or *feature detectors* for different classes of input patterns. In its simplest form, a *competitive neural network*, i.e., a neural network trained using a competitive learning scheme, has a single layer of output neurons that is fully connected. There are also lateral connections among neurons, as illustrated in Figure 4.16, capable of inhibiting or stimulating neighbor neurons.

For a neuron i to be the winner, the distance between its corresponding weight vector $\mathbf{w}_i$ and a certain input pattern $\mathbf{x}$ must be the smallest measure (among all the network output units), given a certain metric $\|\cdot\|$, usually taken to be the Euclidean distance. Therefore, the idea is to find the output neuron whose weight vector is most similar (has the shortest distance) to the input pattern presented.

$$i = \arg\min_i \|\mathbf{x} - \mathbf{w}_i\|, \quad \forall i \tag{4.18}$$

If a neuron does not respond to a determined input pattern (i.e., is not the winner), no learning takes place for this neuron. However, if a neuron i wins the competition, then an adjustment $\Delta\mathbf{w}_i$ is applied to the weight vector $\mathbf{w}_i$ associated with the winning neuron i,

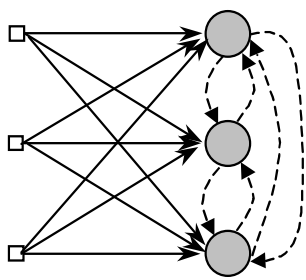


Figure 4.16: Simple competitive network architecture with direct excitatory (feedforward) connections (solid arrows) from the network inputs to the network outputs and inhibitory lateral connections among the output neurons (dashed arrows).

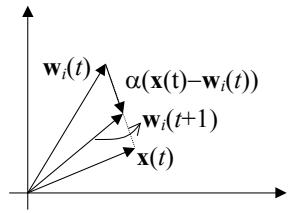


Figure 4.17: Geometric interpretation of the adjustment performed in unsupervised learning. The weight vector of the winning neuron *i* is moved toward the input pattern **x**.

$$\Delta \mathbf{w}_i = \begin{cases} \alpha(\mathbf{x} - \mathbf{w}_i) & \text{if } i \text{ wins the competition} \\ 0 & \text{if } i \text{ loses the competition} \end{cases} \quad \textbf{(4.19)}$$

where α is a *learning rate* that controls the step size given by $\mathbf{w}_i$ in the direction of the input vector **x**. Note that this learning rule works by simply moving the weight vector of the winning neuron in the direction of the input pattern presented, as illustrated in Figure 4.17.

Reinforcement Learning

Reinforcement learning (RL) is distinguished from the other approaches as it relies on learning from direct interaction with the environment, but does not rely on explicit supervision or complete models of the environment. Often, the only information available is a scalar evaluation that indicates how well the artificial neural network is performing. This is based on a framework that defines the interaction between the neural network and its environment in terms of the current values of the network's free parameters (weights), network response (actions), and rewards. Situations are mapped into actions so as to maximize a numerical reward signal (Sutton and Barto, 1998). Figure 4.18 illustrates the network-environment interaction in a reinforcement learning system, highlighting that the network output is fed into the environment that provides it with a reward signal according to how well the network is performing.

In reinforcement learning, the artificial neural network is given a goal to achieve. During learning, the neural network *tries* some actions (i.e., output values) on its environment, then it is *reinforced* or *penalized* by receiving a scalar evaluation (the *reward* or *penalty value*) for its actions. The reinforcement learning algorithm selectively retains the outputs that maximize the received reward over time. The network learns how to achieve its goal by trial-and-error interactions with its environment. At each time step t, the learning system receives some representation of the *state* of the environment $\mathbf{x}(t)$, it provides an output $y(t)$, and one step later it receives a scalar reward $r(t + 1)$, and finds itself in a new state $\mathbf{x}(t + 1)$. Thus, the two basic concepts behind reinforcement learning are *trial and error search* and *delayed reward*.

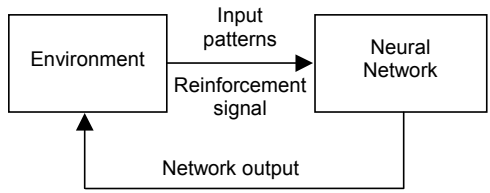


Figure 4.18: In reinforcement learning the network output provides the environment with information about how well the neural network is performing.

The intuitive idea behind reinforcement learning can also be illustrated with a simple example. Assume that you are training in a simulator to become a pilot without any supervisor, and your goal in today's lesson is to land the plane smoothly. Given a certain state of the aircraft, you perform an action that results in the plane crashing on the floor. You thus receive a negative reinforcement (or *punishment*) to that action. If, with the same state of the aircraft as before, you had performed an action resulting in a smooth landing, then you would have received a positive reinforcement (or *reward*). By interacting with the environment and maximizing the amount of reward, you can learn how to smoothly land the plane. We have long been using reinforcement learning techniques to teach animals to play tricks for us, mainly in circuses. For example, consider teaching a dolphin a new trick: you cannot tell it what to do, but you can reward or punish it if it does the right or wrong thing. It has to figure out what it did that made it get the reward or the punishment. This process is also known as the *credit assignment problem*.

4.4 TYPICAL ANNS AND LEARNING ALGORITHMS

The previous section introduced the fundamentals of neurocomputing. It was argued that an artificial neural network can be designed by 1) choosing some abstract models of neurons; 2) defining a network architecture; and 3) choosing an appropriate learning algorithm. In the present section, some of the most commonly used neural networks will be described, focusing on the main three aspects above: type of neuron, network architecture, and learning algorithm. As will be explained throughout this section, the type of learning algorithm is intimately related with the type of application the neural network is going to be used on. In particular, supervised learning algorithms are used for function approximation, pattern classification, control, identification, and other related tasks. On the other hand, unsupervised learning algorithms are employed in data analysis, including clustering and knowledge discovery.

4.4.1. Hebbian Learning

After the 1943 McCulloch and Pitts' paper describing a logical calculus of neuronal activity, N. Wiener published a famous book named *Cybernetics* in 1948, followed by the publication of Hebb's book *The Organization of Behavior*. These were some landmarks in the history of neurocomputing.

In Hebb's book, an explicit statement of a physiological learning rule for synaptic modification was presented for the first time. Hebb proposed that the connectivity of the brain is continually changing as an organism learns distinct functional tasks, and that neural assemblies are created by such changes. Hebb stated that the effectiveness of a variable synapse between two neurons is increased by the repeated activation of one neuron by the other across that synapse. Quoting from Hebb's book *The Organization of Behavior*:

> "When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency as one of the cells firing B, is increased." (Hebb, 1949; p. 62)

This postulate requires that change occur in the synaptic strength between cells when the presynaptic and postsynaptic cells are active at the same time. Hebb suggested that this change is the basis for *associative learning* that would result in a lasting modification in the activity pattern of a spatially distributed assemble of neurons.

This rule is often approximated in artificial neural networks by the *generalized Hebb rule*, where changes in connection strengths are given by the product of presynaptic and postsynaptic activity. The main difference being that change is now a result of both stimulatory and inhibitory synapses, not only excitatory synapses. In mathematical terms,

$$\Delta w_{ij}(t) = \alpha\, y_i(t)\, x_j(t) \qquad \textbf{(4.20)}$$

where Δw_{ij} is the change to be applied to neuron i, α is a *learning rate* parameter that determines how much change should be applied, y_i is the output of neuron i, x_j is the input to neuron i (output of neuron j), and t is the time index. Equation (4.20) can be expressed generically as

$$\Delta w_{ij}(t) = g(y_i(t), x_j(t)) \qquad \textbf{(4.21)}$$

where $g(\cdot,\cdot)$ is a function of both pre- and postsynaptic signals. The weight of a given neuron i is thus updated according to the following rule:

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t) \qquad \textbf{(4.22)}$$

Equation (4.20) clearly emphasizes the correlational or associative nature of Hebb's updating rule. It is known that much of human memory is *associative*, just as the mechanism suggested by Hebb. In an associative memory, an event is linked to another event, so that the presentation of the first event gives rise to the linked event. In the most simplistic version of association, a *stimulus* is linked to a *response*, so that a later presentation of the stimulus evokes the response.

Two important aspects of the Hebbian learning given by Equation (4.20) must be emphasized. First, it is a general updating rule to be used with different types

of neurons. In the most standard case, the Hebbian network is but a single neuron with linear output, but more complex structures could be used. Second, this learning rule is unsupervised; that is, no information about the desired behavior is accounted for. However, the Hebb rule can also be employed when the target output for each input pattern is known. This means that the Hebb rule can also be used in supervised learning. In such cases, the modified Hebb rule has the current output substituted by the desired output:

$$\Delta w_{ij}(t) = \alpha\, d_i(t)\, x_j(t) \qquad \textbf{(4.23)}$$

where, d_i is the desired output of neuron i.

Other variations of the Hebb learning rule take into account, for example, the difference between the neuron's output and its desired output. This leads to the Widrow-Hoff and perceptron learning rules to be described later.

Biological Basis of Hebbian Synaptic Modification

Physiological evidence suggests the existence of Hebb synapses at various locations in the brain. Advances in modern neurophysiological techniques have allowed us to see what appears to be Hebbian modification in several parts of the mammalian brain (Anderson, 1995). A part of the cerebral cortex, the *hippocampus*, shows an effect called *long-term potentiation*, in which its neurons can be induced to display long-term, apparently permanent, increases in activity with particular patterns of stimulation.

Kelso et al. (1986) have presented an experimental demonstration of Hebbian modification using a slice of rat hippocampus maintained outside the animal. They showed that when a presynaptic cell is excited and the postsynaptic cells inhibited, little or no change was seen in the efficiency of a synapse. If the postsynaptic cell was excited by raising the membrane potential at the same time the presynaptic cell was active, the excitatory postsynaptic potential from the presynaptic cell was significantly increased in a stable and long-lasting form.

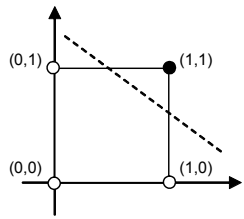
4.4.2. Single-Layer Perceptron

Rosenblatt (1958, 1962) introduced the *perceptron* as the simplest form of a neural network used for the classification of *linearly separable* patterns. Perceptrons constituted the first model of supervised learning, though some perceptrons were self-organized. Before describing the perceptron learning algorithm, the next section discusses in some more detail the problem of linear separability.

Linear Separability

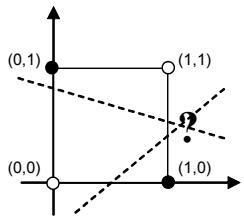
Linear separability can be easily understood with a simple example. Without loss of generality, assume there is a set of input patterns to be classified into a single class. Assume also there is a classifier system that has to respond TRUE if a given input pattern is a member of a certain class, and FALSE if it is not. A TRUE response is represented by an output '1' and a FALSE response by an output '0' of the classifier.

Inputs		Output
x_1	x_2	x_1 AND x_2
0	0	0
0	1	0
1	0	0
1	1	1



(a)

Inputs		Output
x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0



(b)

Figure 4.19: Examples of linear and nonlinear separability. (a) The logic function AND is linearly separable. (b) The logic function XOR is nonlinearly separable.

As one of two responses is required, there is a *decision boundary* that separates one response from the other. Depending on the number m of variables of the input patterns, the decision boundary can have any shape in the space of dimension m. If $m = 2$ and the classifier can be set up so that there is a line dividing all input patterns that produce output '1' from those patterns that produce output '0', then the problem is *linearly separable*; else it is *nonlinearly separable*. This holds true for a space of any dimension, but for higher dimensions, a plane or hyper-plane should exist so as to divide the space into a '1'-class and a '0'-class region.

Figure 4.19 illustrates one linearly separable function and one nonlinearly separable function. Note that, in Figure 4.19(a), several lines could be drawn so as to separate the data into class '1' or '0'. The regions that separate the classes to which each of the input patterns belong to are often called *decision regions*, and the (hyper-)surface that defines these regions are called *decision surfaces*. Therefore, a classification problem can be viewed as the problem of finding decision surfaces that correctly classify the input data. The difficulty lying is that it is not always trivial to automatically define such surfaces.

Simple Perceptron for Pattern Classification

Basically, the perceptron consists of a single layer of neurons with adjustable synaptic weights and biases. Under suitable conditions, in particular if the training patterns belong to linearly separable classes, the iterative procedure of adaptation for the perceptron can be proved to converge to the correct weight set. These weights are such that the perceptron algorithm converges and positions the decision surfaces in the form of (hyper-)planes between the classes. This proof of convergence is known as the *perceptron convergence theorem*.

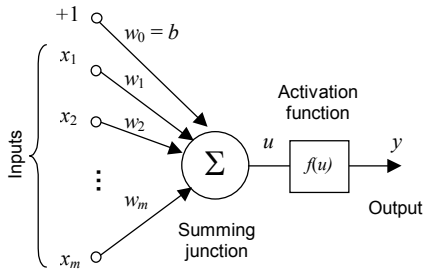


Figure 4.20: The simplest perceptron to perform pattern classification.

The simple perceptron has a single layer of feedforward neurons as illustrated in Figure 4.11(a). The neurons in this network are similar to those of McCulloch and Pitts with a signum or threshold activation function, but include a bias. The key contribution of Rosenblatt and his collaborators was to introduce a learning rule to train the perceptrons to solve pattern recognition and classification problems.

Their algorithm works as follows. For each training pattern $\mathbf{x}_i$, the network output response y_i is calculated. Then, the network determines if an error e_i occurred for this pattern by comparing the calculated output y_i with a *desired value* d_i for that pattern, $e_i = d_i - y_i$. Note that the desired output value for each training pattern is known (supervised learning). The weight vector connecting the inputs (presynaptic neurons) to each output (postsynaptic neuron) and the neuron's bias are updated according to the following rules:

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \alpha\, e_i\, \mathbf{x} \qquad \textbf{(4.24)}$$

$$b(t+1) = b(t) + \alpha\, e \qquad \textbf{(4.25)}$$

where $\mathbf{w} \in \Re^{1\times m}$, $\mathbf{x} \in \Re^{1\times m}$, and $b \in \Re^{1\times 1}$.

Consider now the simplest perceptron case with a single neuron, as illustrated in Figure 4.20. The goal of this network, more specifically neuron, is to classify a number of input patterns as belonging or not belonging to a particular class. Assume that the input data set is given by the N pairs of samples $\{\mathbf{x}_1,d_1\}$, $\{\mathbf{x}_2,d_2\}$, … , $\{\mathbf{x}_N,d_N\}$, where $\mathbf{x}_j$ is the input vector j and d_j its corresponding *target* or *desired* output. In this case, the target value is '1' if the pattern belongs to the class, and '–1' (or '0') otherwise.

Let $\mathbf{X} \in \Re^{m\times N}$, be the matrix of N input patterns of dimension m each (the patterns are placed column wise in matrix $\mathbf{X}$), and $\mathbf{d} \in \Re^{1\times N}$ be the vector of desired outputs. The learning algorithm for this simple perceptron network is presented in Algorithm 4.1, where $f(\cdot)$ is the signum or the threshold function. The stopping criterion for this algorithm is either a fixed number of iteration steps (`max_it`) or the sum E of the squared errors e_i^2, $i = 1, \ldots, N$, for each input pattern being equal to zero. The algorithm returns the weight vector $\mathbf{w}$ as output.

```
procedure [w] = perceptron(max_it,α,X,d)
    initialize w      //set it to zero or small random values
    initialize b      //set it to zero or small random value
    t ← 1; E ← 1
    while t < max_it & E > 0 do,
        E ← 0
        for i from 1 to N do, //for each training pattern
            y_i ← f(w x_i + b)          //network output for x_i
            e_i ← d_i − y_i             //determine the error for x_i
            w ← w + α e_i x_i           //update the weight vector
            b ← b + α e_i               //update the bias term
            E ← E + e_i^2               //accumulate the error
        end for
        t ← t + 1
    end while
end procedure
```

Algorithm 4.1: Simple perceptron learning algorithm. The function $f(\cdot)$ is the signum (or the threshold) function, and the desired output is '1' if a pattern belongs to the class and '–1' (or '0') if it does not belong to the class.

Assuming the input patterns are linearly separable, the perceptron will be capable of solving the problem after a finite number of iteration steps and, thus, the error should be used as the stopping criterion. The parameter α is a *learning rate* that determines the step size of the adaptation in the weight values and bias term. Note, from Algorithm 4.1, that the perceptron learning rule only updates a given weight when the desired response is different from the actual response of the neuron.

Multiple Output Perceptron for Pattern Classification

Note that the perceptron updating rule uses the error-correction learning of most supervised learning techniques, as discussed in Section 4.3.3. This learning rule, proposed to update the weight vector of a single neuron, can be easily extended to deal with networks with more than one output neuron, such as the network presented in Figure 4.11(a). In this case, for each input pattern $\mathbf{x}_i$ the vector of network outputs is given by Equation (4.13) explicitly including the bias vector $\mathbf{b}$ as follows:

$$\mathbf{y} = f(\mathbf{W}\mathbf{x}_i + \mathbf{b}) \tag{4.26}$$

where $\mathbf{W} \in \Re^{o \times m}$, $\mathbf{x}_i \in \Re^{m \times 1}$, $i = 1, \ldots, N$, $\mathbf{y} \in \Re^{o \times 1}$, and $\mathbf{b} \in \Re^{o \times 1}$. The function $f(\cdot)$ is the signum or the threshold function.

Let the matrix of desired outputs be $\mathbf{D} \in \Re^{o \times N}$, where each column of $\mathbf{D}$ corresponds to the desired output for one of the input patterns. Therefore, the error signal for each input pattern $\mathbf{x}_i$ is calculated by simply subtracting the vectors $\mathbf{d}_i$ and $\mathbf{y}_i$ that are of same dimension: $\mathbf{e}_i = \mathbf{d}_i - \mathbf{y}_i$, where $\mathbf{e}_i$, $\mathbf{d}_i$, $\mathbf{y}_i \in \Re^{o \times 1}$. The algorithm to implement the perceptron with multiple output neurons is presented in

```
procedure [W] = perceptron(max_it,α,X,D)
    initialize W     //set it to zero or small random values
    initialize b     //set it to zero or small random values
    t ← 1; E ← 1
    while t < max_it & E > 0 do,
        E ← 0
        for i from 1 to N do,  //for each training pattern
            y_i ← f(W x_i + b)          //network outputs for x_i
            e_i ← d_i − y_i             //determine the error for x_i
            W ← W + α e_i x_i^T         //update the weight matrix
            b ← b + α e_i               //update the bias vector
            E ← E + sum(e_ji^2)         //j = 1,...,o
            ve_i ← e_i
        end for
        t ← t + 1
    end while
end procedure
```

Algorithm 4.2: Learning algorithm for the perceptron with multiple outputs. The function $f(\cdot)$ is the signum (or the threshold) function, and the desired output is '1' if a pattern belongs to the class and '–1' (or '0') if it does not belong to the class. Note that $\mathbf{e}_i$, $\mathbf{d}_i$, $\mathbf{y}_i$, and $\mathbf{b}$ are now vectors and $\mathbf{W}$ is a matrix. e_{ji} corresponds to the error of neuron j when presented with input pattern i.

Algorithm 4.2. As with the simple perceptron, in the perceptron with multiple output neurons the network will converge if the training patterns are linearly separable. In this case, the error for all patterns and all outputs will be zero at the end of the learning phase.

Examples of Application

To illustrate the applicability of the perceptron network, consider the two problems below. The first example - simple classification problem - illustrates the potentiality of the perceptron to represent Boolean functions, and the second example - character recognition - illustrates its capability to recognize a simple set of binary characters.

Simple Classification Problem

Consider the problem of using the simple perceptron with a single neuron to represent the AND function. The training data and its graphical interpretation are presented in Figure 4.21. The training patters to be used as inputs to Algorithm 4.1 are, in matrix notation:

$$\mathbf{X} = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \quad \mathbf{d} = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}$$

Inputs		Output
x_1	x_2	x_1 AND x_2
0	0	0
0	1	0
1	0	0
1	1	1

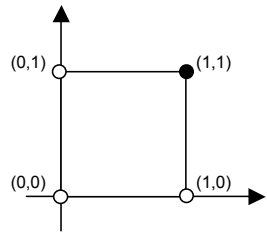


Figure 4.21: The AND function and its graphical representation.

The boundary between the values of x_1 and x_2 for which the network provides a response '0' (not belonging to the class) and the values for which the network responds '1' (belonging to the class) is the separating line given by

$$w_1 x_1 + w_2 x_2 + b = 0 \tag{4.27}$$

Assuming the activation function of the network is the threshold with $\theta = 0$, the requirement for a positive response from the output unit is that the net input it receives be greater than or equal to zero, that is, $w_1 x_1 + w_2 x_2 + b \geq 0$. During training, the values of w_1, w_2, and b are determined so that the network presents the correct response for all training data.

For simplicity, the learning rate is set to 1, $\alpha = 1$, and the initial values for the weights and biases are taken to be zero, $\mathbf{w} = [0\ \ 0]$ and $b = 0$. By running Algorithm 4.1 with $\mathbf{X}$, $\mathbf{d}$, and the other parameters given above, the following values are obtained for the perceptron network with a single output.

$$w_1 = 2;\ w_2 = 1;\ b = -3 \tag{4.28}$$

By replacing these values in the line equation for this neuron we obtain

$$2x_1 + 1x_2 - 3 = 0 \tag{4.29}$$

For each pair of training data, the following net input u_i, $i = 1, \ldots, N$, is calculated: $\mathbf{u} = [-3, -2, -1, 0]$. Therefore, the network response, $\mathbf{y} = f(\mathbf{u})$, for each input pattern is $\mathbf{y} = [0, 0, 0, 1]$, realizing the AND function as desired.

The decision line between the two classes can be determined by isolating the variable x_2 as a function of the other variable in Equation (4.27)

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{b}{w_2} = -2x_1 + 3 \tag{4.30}$$

Note that this line passes exactly on top of class '1', as illustrated in Figure 4.22.

An alternative form of initializing the weights for the perceptron network is by choosing random values within a given domain. For instance, a uniform or normal distribution with zero mean and predefined variance (usually less than or equal to one) could be used. Assume now that the following initial values for $\mathbf{w}$ and b were chosen using a uniform distribution of zero mean and variance one:

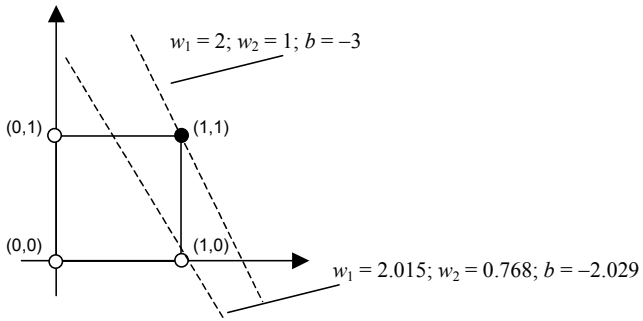


Figure 4.22: Decision lines for the simple perceptron network realizing the logic function AND.

$$w_1 = 0.015; w_2 = 0.768; b = 0.971$$

After training the network, the following values for the weights were determined:

$$w_1 = 2.015; w_2 = 0.768; b = -2.029$$

These values define the decision line below, as depicted in Figure 4.22.

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{b}{w_2} = -2.624\, x_1 + 2.642$$

Two important issues can be noticed here. First, the network response is dependent upon the initial values chosen for its weights and biases. Second, there is not a single correct response for a given training data set. These and other important issues on neural network training will be discussed later.

Character Recognition

As a second example of how to use the perceptron for pattern classification, consider the eight input patterns presented in Figure 4.23. Each input vector is a 120-tuple representing a letter expressed as a pattern on a 12 × 10 grid of pixels. Each character is assumed to have its own class. One output unit is assigned to each character/class, thus there are eight categories to which each character could be assigned. Matrix $\mathbf{X} \in \Re^{120\times 8}$, where each column of $\mathbf{X}$ corresponds to an input pattern, and matrix $\mathbf{D} \in \Re^{8\times 8}$ is a diagonal matrix, meaning that input pattern number 1 (character '0') should activate only output 1, input pattern number 2 (character '1') should activate only output 2, and so on.

Figure 4.23: Training patterns for the perceptron.

```
procedure [y] = run_perceptron(W,b,Z)
    for i from 1 to N do,   //for each training pattern
        y_i ← f(Wz_i + b)       //network outputs for z_i
    end  for
end procedure
```

Algorithm 4.3: Algorithm used to run the trained single-layer perceptron network.

Figure 4.24: Noisy patterns used to test the generalization capability of the single-layer perceptron network trained to classify the patterns presented in Figure 4.23.

After training, the network correctly classifies each of the training patterns. Algorithm 4.3 can be used to run the network in order to evaluate if it is correctly classifying the training patterns. The weight matrix **W** and the bias vector **b** are those obtained after training. Matrix **Z** contains the patterns to be recognized; these can be the original training data **X**, the training data added with noise, or completely new unseen data. Note that, in this case, the network is first trained and then applied to classify novel data; that is, its generalization capability for unseen data can be evaluated.

To test how this network generalizes, some random *noise* can be inserted into the training patterns by simply changing a value '0' by '1' or a value '1' by '0', respectively, with a given probability. Figure 4.24 illustrates the training patterns with 5% noise.

4.4.3. ADALINE, the LMS Algorithm, and Error Surfaces

Almost at the same time Rosenblatt introduced the perceptron learning rule, B. Widrow and his student M. Hoff (Widrow and Hoff, 1960) developed the Widrow-Hoff learning rule, also called the *least mean squared (LMS) algorithm* or *delta rule*. They introduced the ADALINE (ADAptive LInear NEuron) network that is very similar to the perceptron, except that its activation function is linear instead of threshold. Although both networks, ADALINE and perceptron, suffer from only being capable of solving linearly separable problems, the LMS algorithm is more powerful than the perceptron learning rule and has found many more practical uses than the perceptron.

As the ADALINE employs neurons with a linear activation function, the neuron's output is equal to its net input. The goal of the learning algorithm is to minimize the *error* between the network output and the desired output. This allows the network to perform a continuous learning even after a given input pattern has been learnt. One advantage of the LMS algorithm over the perceptron learning rule is that the resultant network after training is not too sensitive to noise. It could be observed in Figure 4.22, that the decision surfaces generated

by the perceptron algorithm usually lie very close to some input data. This is very much the case for the perceptron algorithm, but it is not the case for the LMS algorithm, making the latter more robust to noise. In addition, the LMS algorithm has been broadly used in signal processing applications.

LMS Algorithm (Delta Rule)

The LMS algorithm or delta rule is another example of supervised learning in which the learning rule is provided with a set of inputs and corresponding desired outputs $\{\mathbf{x}_1,\mathbf{d}_1\}$, $\{\mathbf{x}_2,\mathbf{d}_2\}$, …, $\{\mathbf{x}_N,\mathbf{d}_N\}$, where $\mathbf{x}_j$ is the input vector j and $\mathbf{d}_j$ its corresponding *target* or *desired* output vector.

The delta rule adjusts the neural network weights and biases so as to minimize the difference (error) between the network output and the desired output over all training patterns. This is accomplished by reducing the error for each pattern, one at a time, though weight corrections can also be accumulated over a number of training patterns.

The *sum-squared error* (SSE) for a particular training pattern is given by

$$\Im = \sum_{i=1}^{o} e_i^{\,2} = \sum_{i=1}^{o} (d_i - y_i)^2 \qquad \textbf{(4.31)}$$

The gradient of $\Im$, also called the *performance index* or *cost function*, is a vector consisting of the partial derivatives of $\Im$ with respect to each of the weights. This vector gives the direction of most rapid increase of $\Im$; thus, its opposite direction is the one of most rapid decrease in the error value. Therefore, the error can be reduced most rapidly by adjusting the weight w_{IJ} in the following manner

$$w_{IJ} = w_{IJ} - \alpha \frac{\partial \Im}{\partial w_{IJ}} \qquad \textbf{(4.32)}$$

where w_{IJ} is the weight from the J-th presynaptic neuron to the I-th postsynaptic neuron, and α is a learning rate.

It is necessary now to explicitly determine the gradient of the error with respect to the arbitrary weight w_{IJ}. As the weight w_{IJ} only influences the output (postsynaptic) unit I, the gradient of the error is

$$\frac{\partial \Im}{\partial w_{IJ}} = \frac{\partial}{\partial w_{IJ}} \sum_{i=1}^{o} (d_i - y_i)^2 = \frac{\partial}{\partial w_{IJ}} (d_I - y_I)^2$$

It is known that

$$y_I = f(\mathbf{w}_I.\mathbf{x}) = f(\Sigma_j \, w_{Ij}.x_j)$$

Then, we obtain

$$\frac{\partial \Im}{\partial w_{IJ}} = -2(d_I - y_I) \frac{\partial y_I}{\partial w_{IJ}} = -2(d_I - y_I) x_J$$

Therefore, similarly to the updating rule of Hebb's network, the LMS also assumes an updating value Δw_{IJ} to be added to the weight w_{IJ}. The delta rule for

updating the weight from the *J*-th presynaptic neuron to the *I*-th postsynaptic neuron is given by

$$\Delta w_{IJ} = \alpha\ (d_I - y_I)\ x_J \quad \textbf{(4.33)}$$

$$w_{IJ} = w_{IJ} + 2\alpha\ (d_I - y_I)\ x_J \quad \textbf{(4.34)}$$

The following equation for updating the bias b_I can be obtained by calculating the gradient of the error in relation to the bias b_I

$$b_I = b_I + 2\alpha\ (d_I - y_I) \quad \textbf{(4.35)}$$

For each input pattern $\mathbf{x}_i$, the LMS algorithm can be conveniently written in matrix notation as

$$\mathbf{W} = \mathbf{W} + \alpha\ \mathbf{e}_i\ \mathbf{x}_i^T$$

$$\mathbf{b} = \mathbf{b} + \alpha\ \mathbf{e}_i$$

where $\mathbf{W} \in \Re^{o \times m}$, $\mathbf{x}_i \in \Re^{m \times 1}$, $i = 1, \ldots, N$, $\mathbf{e}_i \in \Re^{o \times 1}$, and $\mathbf{b} \in \Re^{o \times 1}$. (Note that 2α was replaced by α.)

The beauty of this algorithm is that at each iteration it calculates an approximation to the gradient vector by simply multiplying the error by the input vector. And this approximation to the gradient can be used in a steepest descent-like algorithm with fixed learning rate.

Error Surfaces

Assume a neural network with *n* weights. This set of weights can be viewed as a point in an *n*-dimensional space, called *weight space*. If the neural network is used to classify a set of patterns, for each of these patterns the network will generate an error signal. This means that every set of weights (and biases) has an associated scalar error value; if the weights are changed, a new error value is determined. The error values for every set of weights define a surface in the weight space, called the *error surface* (Anderson, 1995; Hagan et al., 1996).

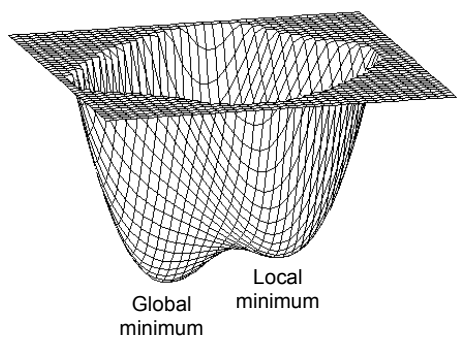


Figure 4.25: An error surface in weight space. The total error is a function of the values of the weights, so every set of weights has an associated error.

The question that arises now is what is the function of learning? The way the LMS algorithm, and the backpropagation algorithm to be explained in the next section, view learning is as the minimization of the error to its smallest value possible. This corresponds to finding an appropriate weight set that leads to the smallest error in the error surface. Figure 4.25 depicts an error surface in weight space. Note that this surface potentially has many *local minima* and one or more *global minimum*. In the example depicted the error surface contains a single local optimum and a single global optimum. Any gradient-like method, such as the LMS and the backpropagation algorithm, can only converge to local optima solutions. Therefore, the choice of an appropriate initial weight set for the network is crucial.

4.4.4. Multi-Layer Perceptron

The multi-layer perceptron is a kind of multi-layer feedforward network such as the one illustrated in Figure 4.12. Typically, the network consists of a set of input units that constitute the input layer, one or more hidden layers, and an output layer. The input signal propagates through the network in a forward direction, layer by layer. This network is a generalization of the single-layer perceptron discussed previously.

In the late 1960's, M. Minsky and S. Papert (1969) released a book called *Perceptrons* demonstrating the limitations of single-layered feedforward networks, namely, the incapability of solving nonlinearly separable problems. This caused a significant impact on the interest in neural network research during the 1970s. Both, Rosenblatt and Widrow were aware of the limitations of the perceptron network and proposed multi-layer networks to overcome this limitation. However, they were unable to generalize their algorithms to train these more powerful networks.

It has already been discussed that a multi-layer network with linear nodes is equivalent to a single-layered network with linear units. Therefore, one necessary condition for the multi-layer network to be more powerful than the single-layer networks is the use of nonlinear activation functions. More specifically, multi-layer networks typically employ a sigmoidal activation function (Figure 4.10).

Multi-layer perceptrons (MLP) have been applied successfully to a number of problems by training them with a supervised learning algorithm known as the *error backpropagation* or simply *backpropagation*. This algorithm is also based on an error-correction learning rule and can be viewed as a generalization of the LMS algorithm or delta rule.

The backpropagation algorithm became very popular with the publication of the *Parallel Distributed Processing* volumes by Rumelhart and collaborators (Rumelhart et al., 1986; McClelland et al., 1986). These books were also crucial for the re-emergence of interest for the research on neural networks. Basically, the error backpropagation learning consists of two passes of computation: a *forward* and a *backward* pass (see Figure 4.12(b)). In the forward pass, an input pattern is presented to the network and its effects are propagated through the

network until they produce the network output(s). In the backward pass, the synaptic weights, so far kept fixed, are updated in accordance with an error correction rule.

The Backpropagation Learning Algorithm

To derive the backpropagation learning algorithm, consider the following notation:

i, j	Indices referring to neurons in the network
t	Iteration counter
N	Number of training patterns
M	Number of layers in the network
$y_j(t)$	Output signal of neuron j at iteration t
$e_j(t)$	Error signal of output unit j at iteration t
$w_{ji}(t)$	Synaptic weight connecting the output of unit j to the input of unit i at iteration t
$u_j(t)$	Net input of unit j at iteration t
$f_j(\cdot)$	Activation function of unit j
$\mathbf{X}$	Matrix of input (training) patterns
$\mathbf{D}$	Matrix of desired outputs
$x_i(t)$	i-th element of the input vector at iteration t
$d_j(t)$	j-th element of the desired output vector at iteration t
α	Learning rate

The description to be presented here follows that of Hagan et al. (1996) and de Castro (1998). For multi-layer networks, the output of a given layer is the input to the next layer,

$$\mathbf{y}^{m+1} = \mathbf{f}^{m+1}(\mathbf{W}^{m+1}\mathbf{y}^{m} + \mathbf{b}^{m+1}),\ m = 0, 1, \ldots, M-1 \qquad \textbf{(4.36)}$$

where M is the number of layers in the network, and the superindex m refers to the layer taken into account (e.g., $m = 0$: input layer, $m = 1$: first hidden layer, … , $m = M - 1$: output layer). The nodes in the input layer ($m = 0$) receive the input patterns

$$\mathbf{y}^{0} = \mathbf{x} \qquad \textbf{(4.37)}$$

that represent the initial condition for Equation (4.36). The outputs in the output layer are the network outputs:

$$\mathbf{y} = \mathbf{y}^{M} \qquad \textbf{(4.38)}$$

Equation (4.14) demonstrates that the network output can be a function of only the input vector $\mathbf{x}$ and the weight matrices $\mathbf{W}^m$. If we explicitly consider the bias terms $\mathbf{b}^m$, Equation (4.13) becomes Equation (4.39) for the network of Figure 4.26:

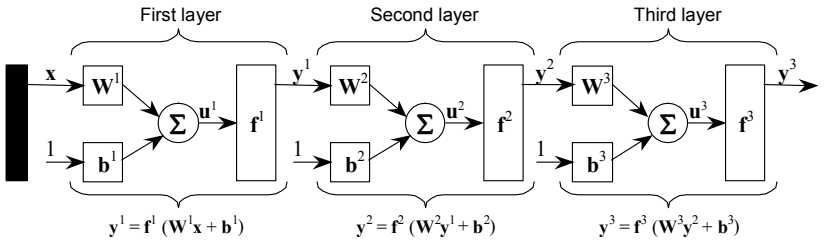


Figure 4.26: An MLP network with three layers. (Matrix notation.)

$$\mathbf{y} = \mathbf{y}^3 = \mathbf{f}^3(\mathbf{W}^3\mathbf{f}^2(\mathbf{W}^2\mathbf{f}^1(\mathbf{W}^1\mathbf{x} + \mathbf{b}^1) + \mathbf{b}^2) + \mathbf{b}^3) \tag{4.39}$$

The backpropagation algorithm to MLP networks is a generalization of the LMS algorithm that uses as a performance index the *mean squared error* (MSE). A set of input patterns and desired outputs is provided, as follows:

$$\{(\mathbf{x}_1,\mathbf{d}_1), (\mathbf{x}_2,\mathbf{d}_2), \ldots, (\mathbf{x}_N,\mathbf{d}_N)\}$$

where $\mathbf{x}_i$ is the i-th input to the network and $\mathbf{d}_i$ is the corresponding desired output, $i = 1,\ldots,N$. From Equation (4.39) it is possible to observe that if ω is the vector of network parameters (weights and biases), then the network output can be given as a function of ω and x:

$$y^M = f(\omega,x)$$

After each training pattern is presented to the network, the network output is compared with the desired output. The algorithm must adjust the vector of parameters so as to minimize the mathematical expectation of the mean squared error:

$$\Im(\omega) = E(e(\omega)^2) = E((d - y(\omega))^2) \tag{4.40}$$

If the network has multiple outputs, Equation (4.40) becomes

$$\Im(\omega) = E(\mathbf{e}(\omega)^T\mathbf{e}(\omega)) = E(\ (\mathbf{d} - \mathbf{y}(\omega))^T\ (\mathbf{d} - \mathbf{y}(\omega))\ )$$

Similarly to the LMS algorithm, the mean squared error can be approximated by the following expression:

$$\hat{\Im}(\omega)\ = \mathbf{e}(t)^T\mathbf{e}(t) = (\mathbf{d}(t) - \mathbf{y}(t))^T\ (\mathbf{d}(t) - \mathbf{y}(t))$$

where the expectation of the mean squared error was substituted by the error at iteration t. In order not to overload the notation, assume $\hat{\Im}(\omega) = \Im(\omega)$.

The updating rule known as steepest descent to minimize the squared error is given by

$$w_{ij}^m(t+1) = w_{ij}^m(t) - \alpha\frac{\partial\Im(t)}{\partial w_{ij}^m} \tag{4.41}$$

$$b_i^m(t+1) = b_i^m(t) - \alpha\frac{\partial\Im(t)}{\partial b_i^m} \tag{4.42}$$

where α is the learning rate.

The most elaborate part is the determination of the partial derivatives that will produce the components of the gradient vector. To determine these derivatives it will be necessary to apply the chain rule a number of times.

The Chain Rule

For a multi-layer network, the error is not a direct function of the weights in the hidden layers, reason why the calculus of these derivatives is not straightforward.

As the error is an indirect function of the weights in the hidden layers, the chain rule must be used to determine the derivatives. The chain rule will be used to determine the derivatives in Equation (4.41) and Equation (4.42).

$$\frac{\partial \Im}{\partial w_{ij}^m} = \frac{\partial \Im}{\partial u_i^m} \times \frac{\partial u_i^m}{\partial w_{ij}^m} \tag{4.43}$$

$$\frac{\partial \Im}{\partial b_i^m} = \frac{\partial \Im}{\partial u_i^m} \times \frac{\partial u_i^m}{\partial b_i^m} \tag{4.44}$$

The second term of both equations above can be easily determined for the net input of layer *m* as an explicit function of the weights and biases in this layer:

$$u_i^m = \sum_{j=1}^{S^{m-1}} w_{ij}^m y_j^{m-1} + b_i^m \tag{4.45}$$

where S^m is the number of neurons in layer *m*.

Therefore,

$$\frac{\partial u_i^m}{\partial w_{ij}^m} = y_j^{m-1}, \frac{\partial u_i^m}{\partial b_i^m} = 1 \tag{4.46}$$

Now define the *sensitivity* (δ) of $\Im$ to changes in the *i*-th element of the net input in layer *m* as

$$\delta_i^m \equiv \frac{\partial \Im}{\partial u_i^m} \tag{4.47}$$

Equation (4.43) and Equation (4.44) can now be simplified by

$$\frac{\partial \Im}{\partial w_{ij}^m} = \delta_i^m y_j^{m-1} \tag{4.48}$$

$$\frac{\partial \Im}{\partial b_i^m} = \delta_i^m \tag{4.49}$$

Thus, Equation (4.41) and Equation (4.42) become

$$w_{ij}^m(t+1) = w_{ij}^m(t) - \alpha \delta_i^m y_j^{m-1} \tag{4.50}$$

$$b_i^m(t+1) = b_i^m(n) - \alpha\delta_i^m \tag{4.51}$$

In matrix notation these equations become

$$\mathbf{W}^m(t+1) = \mathbf{W}^m(t) - \alpha\boldsymbol{\delta}^m(\mathbf{y}^{m-1})^T \tag{4.52}$$

$$\mathbf{b}^m(t+1) = \mathbf{b}^m(t) - \alpha\boldsymbol{\delta}^m \tag{4.53}$$

where

$$\boldsymbol{\delta}^m \equiv \frac{\partial \Im}{\partial \mathbf{u}^m} = \begin{bmatrix} \dfrac{\partial \Im}{\partial u_1^m} \\ \dfrac{\partial \Im}{\partial u_2^m} \\ \vdots \\ \dfrac{\partial \Im}{\partial u_{S^m}^m} \end{bmatrix} \tag{4.54}$$

Backpropagating the Sensitivities

It is now necessary to calculate the sensitivity $\boldsymbol{\delta}^m$ that requires another application of the chain rule. This process gives rise to the term *backpropagation* because it describes a recurrence relationship in which the sensitivity of layer m is determined from the sensitivity of layer $m + 1$.

To derive the recurrence relationship for the sensitivities, we will use the following Jacobian matrix

$$\frac{\partial \mathbf{u}^{m+1}}{\partial \mathbf{u}^m} = \begin{bmatrix} \dfrac{\partial u_1^{m+1}}{\partial u_1^m} & \dfrac{\partial u_1^{m+1}}{\partial u_2^m} & \cdots & \dfrac{\partial u_1^{m+1}}{\partial u_{S^m}^m} \\ \dfrac{\partial u_2^{m+1}}{\partial u_1^m} & \dfrac{\partial u_2^{m+1}}{\partial u_2^m} & \cdots & \dfrac{\partial u_2^{m+1}}{\partial u_{S^m}^m} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial u_{S^{m+1}}^{m+1}}{\partial u_1^m} & \dfrac{\partial u_{S^{m+1}}^{m+1}}{\partial u_2^m} & \cdots & \dfrac{\partial u_{S^{m+1}}^{m+1}}{\partial u_{S^m}^m} \end{bmatrix} \tag{4.55}$$

Next, we want to find an expression for this matrix. Consider the element i,j of this matrix

$$\frac{\partial u_i^{m+1}}{\partial u_j^m} = w_{ij}^{m+1}\frac{\partial y_j^m}{\partial u_j^m} = w_{ij}^{m+1}\frac{\partial f^m(u_j^m)}{\partial u_j^m} = w_{ij}^{m+1}\dot{f}^m(u_j^m) \tag{4.56}$$

where

$$\dot{f}^m(u_j^m) = \frac{\partial f^m(u_j^m)}{\partial u_j^m} \tag{4.57}$$

Therefore, the Jacobian matrix can be written as

$$\frac{\partial \mathbf{u}^{m+1}}{\partial \mathbf{u}^{m}} = \mathbf{W}^{m+1}\dot{\mathbf{F}}^{m}(\mathbf{u}^{m}) \tag{4.58}$$

where

$$\dot{\mathbf{F}}^{m}(\mathbf{u}^{m}) = \begin{bmatrix} \dot{f}^{m}(u_1^m) & 0 & \cdots & 0 \\ 0 & \dot{f}^{m}(u_2^m) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \dot{f}^{m}(u_{S^m}^m) \end{bmatrix} \tag{4.59}$$

It is now possible to write the recurrence relationship for the sensitivity using the chain rule in matrix form

$$\begin{aligned} \boldsymbol{\delta}^{m} &= \frac{\partial \Im}{\partial \mathbf{u}^{m}} = \left(\frac{\partial \mathbf{u}^{m+1}}{\partial \mathbf{u}^{m}}\right)^{T} \frac{\partial \Im}{\partial \mathbf{u}^{m+1}} = \dot{\mathbf{F}}^{m}(\mathbf{u}^{m})(\mathbf{W}^{m+1})^{T} \frac{\partial \Im}{\partial \mathbf{u}^{m+1}} \\ &= \dot{\mathbf{F}}^{m}(\mathbf{u}^{m})(\mathbf{W}^{m+1})^{T}\boldsymbol{\delta}^{m+1} \end{aligned} \tag{4.60}$$

Note that the sensitivities are (back)propagated from the last to the first layer

$$\boldsymbol{\delta}^{M} \rightarrow \boldsymbol{\delta}^{M-1} \rightarrow \ldots \rightarrow \boldsymbol{\delta}^{2} \rightarrow \boldsymbol{\delta}^{1} \tag{4.61}$$

There is a last step to be executed to complete the backpropagation algorithm. We need the starting point, $\boldsymbol{\delta}^{M}$, for the recurrence relation of Equation (4.60).

$$\delta_i^{M} = \frac{\partial \Im}{\partial u_i^{M}} = \frac{\partial (\mathbf{d}-\mathbf{y})^{T}(\mathbf{d}-\mathbf{y})}{\partial u_i^{M}} = \frac{\partial \sum_{j=1}^{S^M}(d_j - y_j)^2}{\partial u_i^{M}} = -2(d_i - y_i)\frac{\partial y_i}{\partial u_i^{M}} \tag{4.62}$$

Now, since

$$\frac{\partial y_i}{\partial u_i^{M}} = \frac{\partial y_i^{M}}{\partial u_i^{M}} = \frac{\partial f^{M}(u_j^{M})}{\partial u_i^{M}} = \dot{f}^{M}(u_j^{M}) \tag{4.63}$$

It is possible to write

$$\delta_i^{M} = -2(d_i - y_i)\dot{f}^{M}(u_j^{M}) \tag{4.64}$$

Or equivalently in matrix notation

$$\boldsymbol{\delta}^{M} = -2\dot{\mathbf{F}}^{M}(\mathbf{u}^{M})(\mathbf{d}-\mathbf{y}) \tag{4.65}$$

Figure 4.27 provides an adaptation, using matrix notation, of the result presented by Narendra and Parthasarathy (1990) and describes a flow graph for the backpropagation algorithm.

Algorithm 4.4 presents the pseudocode for the standard backpropagation algorithm. The value of α should be small, $\alpha \in (0,1]$. As this algorithm updates the weight matrices of the network after the presentation of each input pattern, this could cause a certain bias toward the order in which the training patterns are presented. In order to avoid this, the patterns are presented to the network in a random order, as described in Algorithm 4.4.

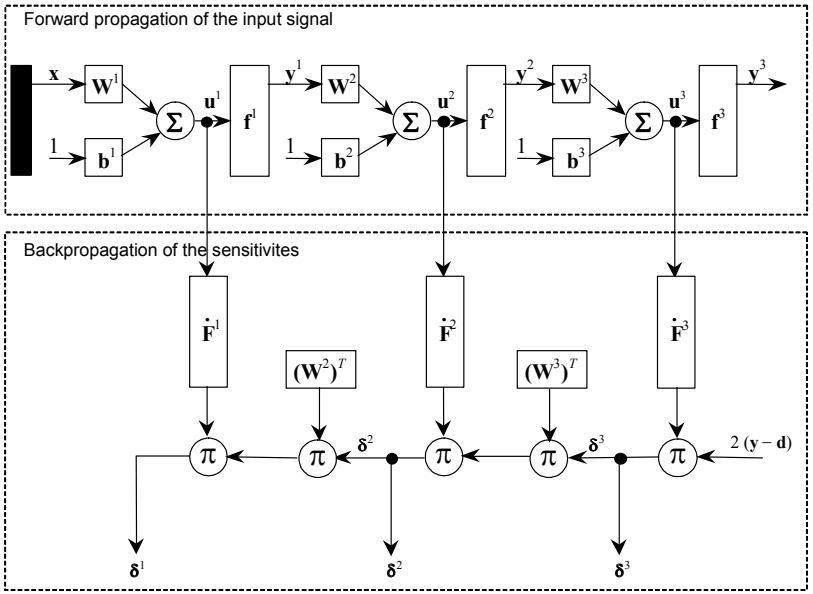


Figure 4.27: Architectural graph of an MLP network with three layers representing the forward propagation of the functional signals and the backward propagation of the sensitivities.

```
procedure [W] = backprop(max_it,min_err,α,X,D)
    for m from 1 to M do,
        initialize W^m //small random values
        initialize b^m //small random values
    end for
    t ← 1
    while t < max_it & MSE > min_err do,
        vet_permut ← randperm(N)  //permutations of N
        for j from 1 to N do,      //for all input patterns
            //select the index i of pattern x_i to be presented
            i ← vet_permut(j)  //present patterns randomly
            //forward propagation of the functional signal
            y^0 ← x_i                                  //Equation (4.37)
            for m from 0 to M - 1 do,
                y_i^{m+1} ← f^{m+1}(W^{m+1} y_i^m + b^{m+1})   //Equation (4.36)
            end for
            //backpropagation of sensitivities
            δ_i^M ← -2Ḟ^M(u_i^M)(d_i - y_i)            //Equation (4.65)
            for m from M - 1 down to 1 do,
                δ_i^m ← Ḟ^m(u_i^m)(W^{m+1})^T δ_i^{m+1},   //Equation (4.60)
            end for
            //update weights and biases
            for m from 1 to M do,
                W^m ← W^m - αδ_i^m (y_i^{m-1})^T,      //Equation (4.52)
                b^m ← b^m - αδ_i^m,                    //Equation (4.53)
            end for
            //calculate the error for pattern i
            E_i ← e_i^T e_i = (d_i - y_i)^T (d_i - y_i)
        end for
        MSE ← 1/N.sum(E_i)                             //Mean Square Error
        t ← t + 1
    end while
end procedure
```

Algorithm 4.4: Learning algorithm for the MLP network trained via the backpropagation algorithm.

Universal Function Approximation

An MLP network can be seen as a generic tool to perform a *nonlinear input-output mappings*. More specifically, let m be the number of inputs to the network and o the number of outputs. The input-output relationship of the network defines a mapping from an input m-dimensional Euclidean space into an output o-dimensional Euclidean space, which is infinitely continuously differentiable.

Cybenko (1989) was the first researcher to rigorously demonstrate that an MLP neural network with a single hidden layer is sufficient to uniformly approximate any continuous function that fits a unit hypercube.

The *universal function approximation theorem* is as follows:

Theorem: Let $f(\cdot)$ be a nonconstant continuous, limited, and monotonically increasing function. Let $\boldsymbol{I}_m$ be a unit hypercube of dimension m, $(0,1)^m$. The space of continuous functions in $\boldsymbol{I}_m$ is denoted by $\boldsymbol{C}(\boldsymbol{I}_m)$. Thus, given any function $g \in \boldsymbol{C}(\boldsymbol{I}_m)$ and $\varepsilon > 0$, there is an integer M and sets of real-valued constants α_i and w_{ij}, where $i = 1, \ldots, o$ and $j = 1, \ldots, m$, it is possible to define

$$F(x_1, x_2, ..., x_m) = \sum_{i=1}^{o} \alpha_i f\left(\sum_{j=1}^{m} w_{ij} x_j - w_{i0} \right) \qquad \textbf{(4.66)}$$

as an approximation to the function $g(.)$ such that

$|F(x_1, \ldots, x_m) - g(x_1, \ldots, x_m)| < \varepsilon$ for all $\{x_1, \ldots, x_m\} \in \boldsymbol{I}_m$.

Proof: see Cybenko (1989).

This theorem is directly applicable to the MLP networks. First, note that the sigmoidal function used in the MLP networks is continuous, nonconstant, limited, and monotonically increasing; satisfying the constraints imposed to $f(\cdot)$. Then, note that Equation (4.66) represents the outputs of an MLP network, as illustrated in Figure 4.28.

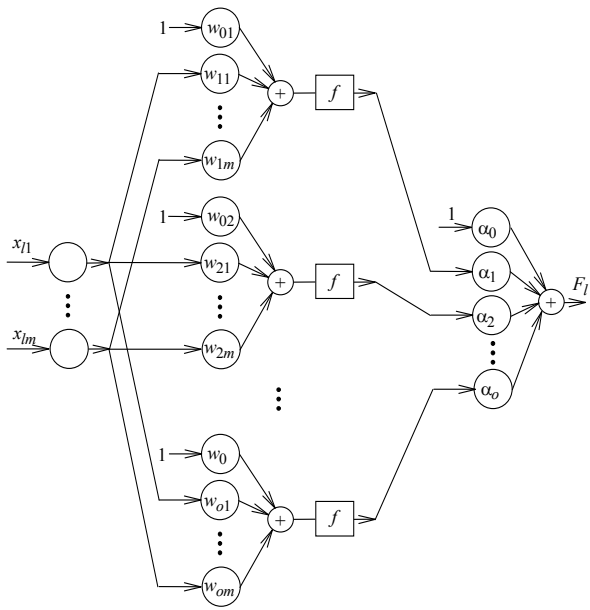


Figure 4.28: An MLP network as a universal function approximator. The network parameters compose Equation (4.66). (Courtesy of © Fernando J. Von Zuben.)

In summary, the theorem states that an MLP network with a single hidden layer is capable of uniform approximation, given an appropriate training data set to represent the function. However, the theorem does not say anything regarding the number of units in the hidden layer necessary to perform an approximation with the precision ε.

Some Practical Aspects

There are a several aspects of the MLP network training that deserve some comments. Some of these aspects are also valid for the other networks presented and to be presented in this chapter.

- *Network architecture*: the number of network inputs is usually defined by the training data, while the number of outputs and hidden units are design issues. For instance, if a data set to be classified has a single class, one or two output units can be used. In the case of a single sigmoidal output in the range (–1,1), a network output less than '0' can be considered as not belonging to the class, and an output greater than or equal to zero can be considered as belonging to the class. If two output units are used, then one of them corresponds to belonging to the class, while the other corresponds to not belonging to the class. Finally, the number of hidden units is chosen so that the network can appropriately classify or approximate the training set. By increasing the number of hidden units, one increases the network mapping capability. It is important to have in mind that an excessive number of hidden units may result in overfitting, while a small number may lead to underfitting.
- *Generalization*: we have already discussed that too much training may result in overfitting, while too little training may result in underfitting. Overfitting is only possible if there are enough hidden units to promote an increasingly better representation of the data set. On the other side, if the network is not big enough to perform an adequate input-output mapping, underfitting may occur.
- *Convergence*: the MLP network is usually trained until a minimum value for the error is achieved; for instance, stop training if the MSE < 0.001. Another choice for the *stopping criterion* is to stop training if the estimated gradient vector at iteration t has a small norm; for instance, stop training if $\|\partial\Im/\partial w\| < 0.001$. A small value for the norm of the gradient value indicates that the network is close to a local minimum.
- *Epoch*: after all the training patterns have been presented to the network and the weights adjusted, an *epoch* is said to have been completed. This terminology is used in all artificial neural network learning algorithms, including the perceptron and self-organizing maps.
- *Data normalization*: when the activation functions of the neurons have a well-defined range, such as all sigmoidal functions, it is important to normalize the data such that they belong to a range equal to or smaller than the range of the activation functions. This helps to avoid the saturati-

on of the neurons in the network. Normalizing the training data is also important when their attributes range over different scales. For example, a data set about cars may include variables such as color (symbolic or discrete attribute), cost (real-valued attribute), type (off road, van, etc.), and so on. Each of these attributes assumes a value on a different range and, thus, influences the network with different degrees. By normalizing all attributes to the same range, we reduce the importance of the differences in scale.

- *Initialization of the weight vectors and biases*: most network models, mainly multi-layer feedforward networks, are sensitive to the initial values chosen for the network weights. This sensitivity can be expressed in several ways, such as guarantee of convergence, convergence speed, and convergence to local optima solutions. Usually, small random values are chosen to initialize the network weights.
- *Learning rate*: the learning rate α plays a very important role in neural network training because it establishes the size of the adaptation step to be performed on a given weight. Too small learning rates may result in a very long learning time, while too large learning rates may result in instability and nonconvergence.

Biological Plausibility of Backpropagation

The backpropagation of error signals is probably the most problematic feature in biological terms. Despite the apparent simplicity and elegance of the backpropagation algorithm, it seems quite implausible that something like the equations described above are computed in the brain (Anderson, 1995; O'Reilly and Munakata, 2000; Trappenberg, 2002).

In order to accomplish this, some form of information exchange between postsynaptic and presynaptic neurons should be possible. For instance, there would be the requirement, in biological terms, that the sensitivity values were propagated backwards from the dendrite of the postsynaptic neuron, across the synapse, into the axon terminal of the presynaptic neuron, down the axon of this neuron, and then integrated and multiplied by both the strength of that synapse and some kind of derivative, and then propagated back out its dendrites, and so on. If a neuron were to gather the backpropagated errors from all the other nodes to which it projects, some synchronization issues would arise, and it would also affect the true parallel processing in the brain.

However, it is possible to rewrite the backpropagation equations so that the error backpropagation between neurons takes place using standard activation signals. This approach has the advantage that it ties into the psychological interpretation of the teaching signal (desired output) as an actual state of experience that reflects something like an outcome or corrected response. The result is a very powerful learning algorithm that need not ignore issues of biological plausibility (Hinton and McClelland, 1987; O'Reilly, 1996).

Examples of Application

Two simple applications for the MLP network will be presented here. The first problem - universal function approximation - aims at demonstrating that the MLP network with nonlinear hidden units is capable of universal approximation, and how this is accomplished. The second example - design of a controller - is a classic example from the literature that demonstrates the capability of using an MLP network trained with error backpropagation to implement a self-learning controller.

Universal Function Approximation

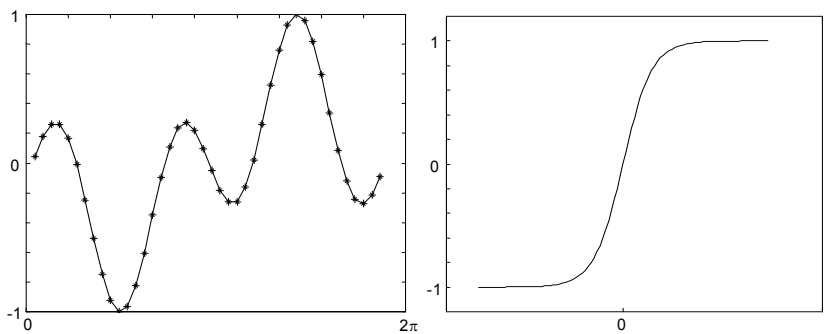
To illustrate the universal function approximation of the MLP neural networks, consider the following example: approximate one period of the function $sin(x).cos(2x)$ using an additive composition of basic functions ($f(\cdot)$ is the hyperbolic tangent function) under the form of Equation (4.66), as illustrated in Figure 4.29.

The MLP network depicted in Figure 4.30 is used to approximate this function. The architecture is: $m = 1$ input, $S^1 = 5$ hidden units in a single hidden layer, and $o = 1$ output unit. The training algorithm is the backpropagation introduced above. By looking at this picture, and assuming the network has a linear output, y is expressed as

$$y = w_{10} + w_{11} z_1 + w_{12} z_2 + w_{13} z_3 + w_{14} z_4 + w_{15} z_5 \qquad \textbf{(4.67)}$$

where

$$z_l = f\left(\sum_{j=1}^{m} v_{lj} x_j + v_{l0} \right), \quad l = 1, \ldots, 5 \qquad \textbf{(4.68)}$$



(a) (b)

Figure 4.29: Universal approximation. (a) Function to be approximated (information available: 42 training data equally sampled in the domain of the function. (b) Basic function to be used in the approximation: hyperbolic tangent, satisfying the constraints of the universal function approximation theorem.

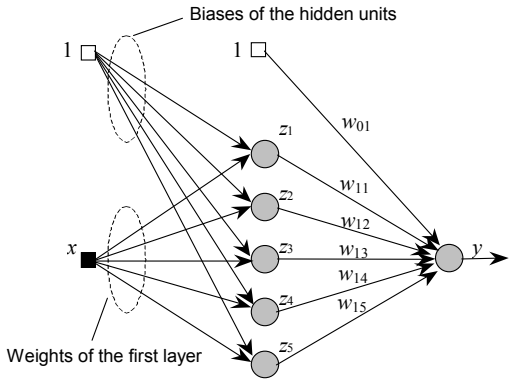


Figure 4.30: MLP network trained via the error backpropagation algorithm used to approximate the function $sin(x).cos(2x)$. The activation functions are hyperbolic tangents.

where $f(\cdot)$ is the hyperbolic tangent, m is the number of network inputs, v_{lj} is the weight of the synapse connecting input j to the hidden unit l. (Compare Equation (4.67) and Equation (4.68) with Equation (4.66).)

After training the network so as to define appropriate weight sets, let us analyze the components of Equation (4.67) that determine the network output. Figure 4.31 depicts the activation functions (basic functions) of the hidden units after network training. Note that these curves are hyperbolic tangents translated or scaled along the axes. Figure 4.32 presents the linear combination of the outputs of the hidden units, where the coefficients of the combination are the weights of the second layer in the network. Finally, the resultant approximation is presented in Figure 4.33.

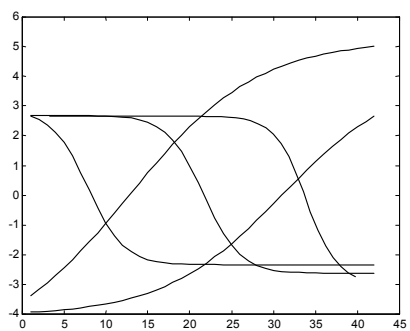


Figure 4.31: Activation functions of the hidden units, after the network training, multiplied by the corresponding weights of the output layer. Note that all functions are hyperbolic tangents (basic functions).

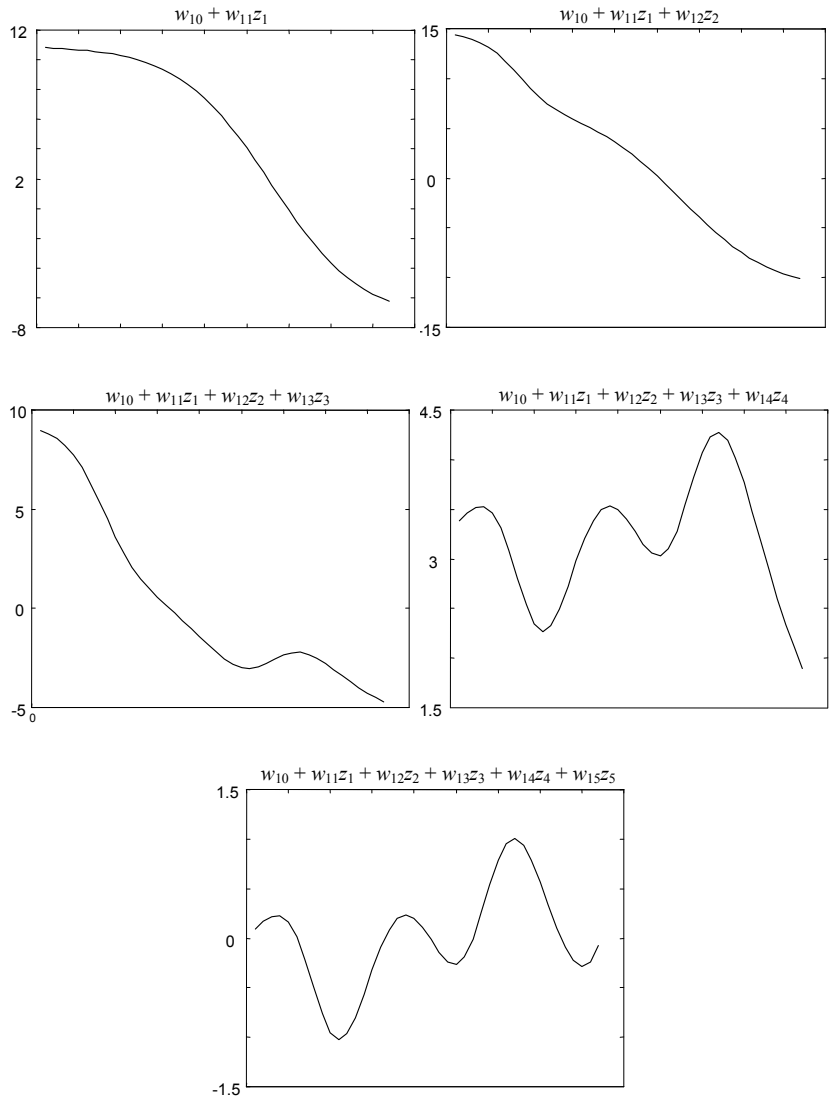


Figure 4.32: Linear combination of the outputs of the hidden units. The coefficients of the linear combination are the weights of the second layer. This figure presents the sum of the curves that compose Figure 4.31, corresponding to the additive composition (linear combination) of the basic functions of the network's neurons.

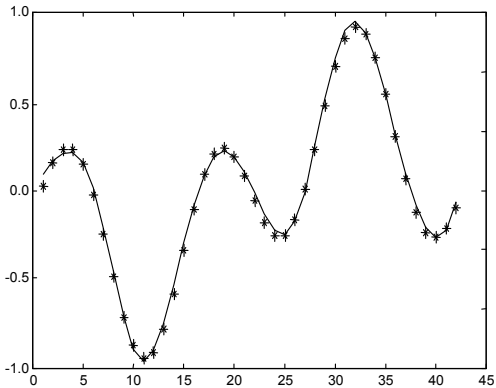


Figure 4.33: Approximation obtained by the MLP network for the function $sin(x).cos(2x)$. The '*' correspond to the training samples, and the line (—) is the network approximation.

Design of a Controller

In a classic paper from the ANN literature, Nguyen and Widrow (1990) applied a simple multi-layer perceptron network trained with the standard error back-propagation algorithm to design a neuro-controller to control the steering of a trailer truck while backing up to a loading dock from any initial position. The authors wanted to demonstrate how a simple neural network architecture could be used to implement a self-learning nonlinear controller for the truck when only backing up was allowed. The network architecture was a standard MLP with a single hidden layer, 25 hyperbolic tangents in the hidden layer and a single sigmoid (tanh) output unit.

The critical state variables representing the position of the truck and that of the loading dock are the angle of the cab θ_{cab}, the angle of the trailer $\theta_{trailer}$, and the Cartesian position of the rear of the center of the trailer $(x_{trailer}, y_{trailer})$. The truck is placed at some initial position and is backed up while being steered by the controller, and the run ends when the truck gets to the dock, as illustrated in Figure 4.34. The goal is to cause the back of the trailer to be parallel to the loading dock; that is, $\theta_{trailer} = 0$ and to have $(x_{trailer}, y_{trailer})$ as close as possible to (x_{dock}, y_{dock}). The final cab angle is not important. An objective function that allows the controller to learn to achieve these objectives by adapting the network weights and biases is:

$$\Im = E[\alpha_1(x_{dock} - x_{trailler})^2 + \alpha_2(y_{dock} - y_{trailler})^2 + \alpha_3(0 - \theta_{trailler})^2]$$

where the constants α_1, α_2 and α_3 are chosen by the user so as to weigh the importance of each error component, and $E[\cdot]$ corresponds to the average (expectation) over all training runs.

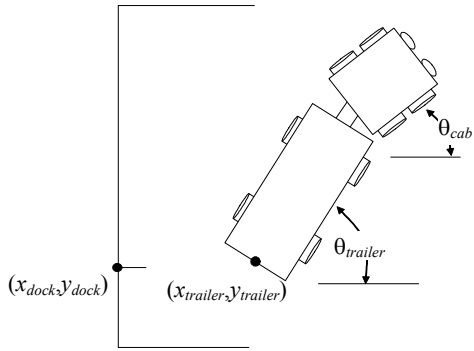


Figure 4.34: Illustration of the truck, trailer, loading dock, and the parameters involved. (Adapted with permission from [Nguyen and Widrow, 1990], © IEEE Press.)

The training of the controller was divided into several 'lessons'. In the beginning, the controller was trained with the truck initially very close to the dock and the trailer pointing at the dock. Once the controller was proficient at working with these initial conditions, the problem was made harder by placing the truck farther away from the dock and with increasingly more difficult angles. This way, the controller learned how to deal with easy situations first, and then was adapted to deal with harder scenarios.

The MLP neural network trained with the standard backpropagation of errors learning algorithm demonstrated to be capable of solving the problem by controlling the truck very well. The truck and trailer were placed in various different initial positions, and backing up was performed successfully in each case. The trained controller was capable of controlling the truck from initial positions it had never seen before, demonstrating the generalization capability of the neural network studied.

4.4.5. Self-Organizing Maps

The *self-organizing map* (SOM) discussed in this section is an unsupervised (self-organized or competitive; Section 4.3.3) network introduced by T. Kohonen (1982). The network architecture is a single layer feedforward network with the postsynaptic neurons placed at the nodes of a *lattice* or *grid* that is usually uni- or bi-dimensional, as illustrated in Figure 4.35. This network is trained via a competitive learning scheme, thus there are lateral connections between the output units which are not shown in the picture. (See Appendix B.4.5 for a brief introduction to data clustering.)

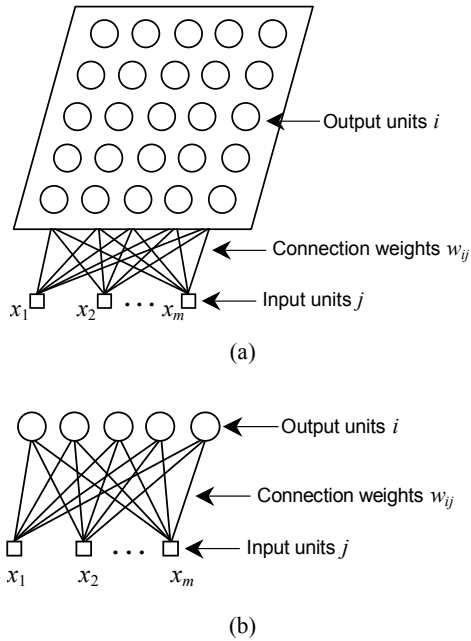


Figure 4.35: Typical architectures (single layer feedforward) of a self-organized map. (a) Bi-dimensional output grid. (b) Uni-dimensional output grid.

The output units become selectively tuned to various input signal patterns or clusters of patterns through an unsupervised learning process. In the standard version of the SOM to be presented here, only one neuron or local group of neurons at a time gives the active response to the current input. The locations of the responses tend to become ordered as if some meaningful coordinate system for different input features were being created over the network (Kohonen, 1990). A self-organizing map is thus characterized by the formation of a topographic map of the input patterns in which the spatial location or coordinates of a neuron in the network are indicative of intrinsic statistical features contained in the input patterns.

For instance, Figure 4.36 depicts a possible synaptic weight distribution for a uni-dimensional self-organizing map with two inputs, $m = 2$. Note that, although the output array allows that the neurons can be freely positioned in the space of the input data, its dimension is still one. Thus, the interpretation of this uni-dimensional output array, after self-organization, occurs in a uni-dimensional space independently of the original data dimension.

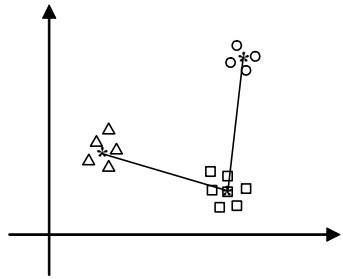


Figure 4.36: Possible distribution of weight vectors for two-input patterns and a uni-dimensional array of neurons in a SOM. The black stars correspond to the position of the SOM weight vectors and the lines connecting them represent their neighborhood relationships. Triangles, rectangles, and circles correspond to different clusters of data that are presented to the network without their labels; that is, the network is not informed that they belong to different groups.

Self-Organizing Map Learning Algorithm

The main objective of the self-organizing map is to transform an input pattern of dimension *m* into a uni- or bi-dimensional discrete map, and to perform this transformation adaptively in a topologically ordered form. Each unit in the grid presented in Figure 4.35 is fully connected to all the presynaptic units in the input layer. The uni-dimensional array of Figure 4.35(b) is a particular case of the bi-dimensional grid of Figure 4.35(a).

The self-organizing learning algorithm proceeds by first initializing the synaptic strengths of the neurons in the network, what usually follows the same rule of the other networks discussed; that is, by assigning them small values picked from a random or uniform distribution. This stage is important so that no *a priori* order is imposed into the map. After initialization, there are three essential processes involved in the formation of the self-organized map (Haykin, 1999):

- *Competition*: for each input pattern the neurons in the network will compete with one another in order to determine the winner.
- *Cooperation*: the winning neuron determines the spatial location (neighborhood) around which other neighboring neurons will also be stimulated. It is crucial to the formation of ordered maps that the cells doing the learning are not affected independently of one another, but as topologically related subsets, on each of which a similar kind of correction is imposed.
- *Adaptation*: the winning neuron and its neighbors will have their associated weight vectors updated. Usually the degree of adaptation of the weight vectors is proportional to their distance, in the topological neighborhood, to the winner: the closer to the winner, the higher the degree of adaptation, and vice-versa.

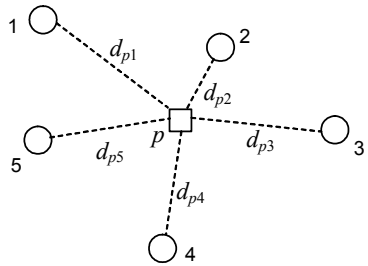


Figure 4.37: Competitive phase.

Let $\mathbf{x} = [x_1, x_2, \ldots, x_m]^T \in \Re^m$ be an input vector (training pattern) that is assumed to be presented in parallel to all the neurons $i = 1, \ldots, o$, in the network. The whole data set $\mathbf{X}$ is composed on N patterns, $\mathbf{X} \in \Re^{m \times N}$. The weight vector of unit i is $\mathbf{w}_i = [w_{i1}, w_{i2}, \ldots, w_{im}] \in \Re^{1 \times m}$, and thus the weight matrix of the network is $\mathbf{W} \in \Re^{o \times m}$, where o is the number of output (postsynaptic) units in the network.

Competition

For each input pattern, the o neurons in the network will compete to become activated. One simple way of performing the competition among the output units is by applying the competitive learning scheme discussed in Section 4.3.3. That is, for a neuron i to be the winner, the distance between its corresponding weight vector $\mathbf{w}_i$ and a certain input pattern $\mathbf{x}$ must be the smallest measure (among all the network output units), given a certain metric $\| \cdot \|$, usually taken to be the Euclidean distance. Let $i(\mathbf{x})$ indicate the winner neuron for input pattern $\mathbf{x}$. It can thus be determined by

$$i(\mathbf{x}) = \arg\min_j \|\mathbf{x} - \mathbf{w}_j\|, \; j = 1, \ldots, o \qquad \textbf{(4.69)}$$

Neuron $i(\mathbf{x})$ that satisfies Equation (4.69) is known as the best-matching, winner, or winning unit for the input pattern $\mathbf{x}$.

Figure 4.37 illustrates the competition in a SOM. Assume a map with 5 neurons and an input pattern p presented to the network. Each value d_{pj}, $j = 1, \ldots, 5$, corresponds to the Euclidean distance between the neuron weight vector and the input pattern p. In this example, the winner neuron is number 2, because it has the smallest Euclidean distance to the input pattern and is, thus, the neuron most activated by the input pattern p.

Cooperation

The winning neuron locates the center of a topological neighborhood of cooperating neurons. There are neurobiological evidences for the lateral interaction among a set of excited neurons. In particular, a neuron that is firing tends to excite the neurons in its immediate neighborhood more than those far away from it. One form of simulating this is by defining topological neighborhoods around a winning neuron, in which a similar kind of adaptation to that of the winner will be imposed.

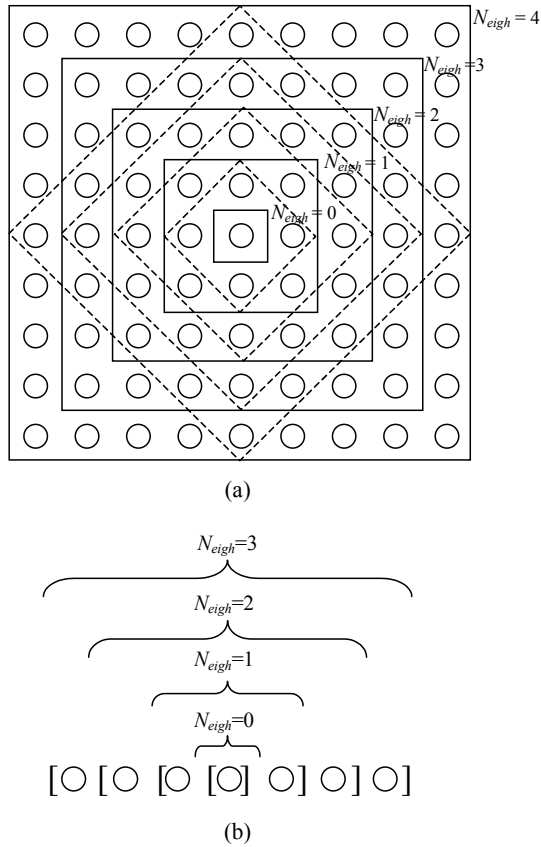


Figure 4.38: Examples of topological neighborhoods for bi-dimensional (a) and uni-dimensional grids (b).

In biophysically inspired neural networks, there are a number of ways of defining the topological neighborhood by using various kinds of lateral feedback connections and other lateral interactions. In the present discussion, lateral interaction is enforced directly by defining a neighborhood set N_{eigh} around a winning unit u. At each iteration, all the units within N_{eigh} are updated, whereas units outside N_{eigh} are left intact. The width or radius of N_{eigh} is usually time variant; N_{eigh} is initialized very wide in the beginning of the learning process and shrinks monotonically with time. Figure 4.38 illustrates some typical topological neighborhoods for bi-dimensional and uni-dimensional output grids. In Figure 4.38(a), two different forms of implementing a squared neighborhood for a bi-dimensional grid are depicted. Note that other neighborhoods are still possible, such as a hexagonal (or circular) neighborhood.

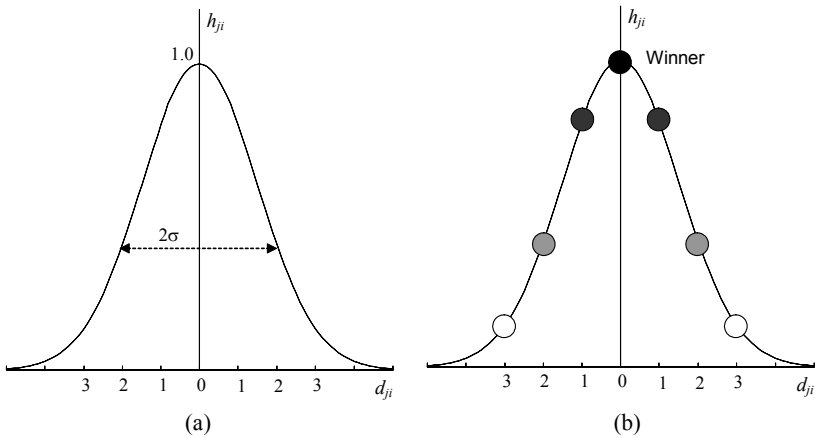


Figure 4.39: Neighborhood. (a) Gaussian neighborhood. (b) Neighborhood relationships: gray levels indicate the neighborhood degree of each unit. The winner neuron has a higher value for h, $h_{ji(\mathbf{x})}$, while h_{ji} of the other neurons decreases for larger distances from the winner. (Courtesy of © Lalinka C. T. Gomes)

Another form of simulating the influence of a neuron in its immediate neighborhood is to define a general kernel function $h_{ji(\mathbf{x})}$ that allows a decay of the neurons updating smoothly with lateral distance (Figure 4.39). One form of implementing this is by giving function $h_{ji(\mathbf{x})}$ a "bell curve" shape. More specifically, let $h_{ji(\mathbf{x})}$ denote the topological neighborhood centered on the winning neuron $i(\mathbf{x})$ and encompassing a set of neighbor neurons $j \in N_{eigh}$, and let $d_{i(\mathbf{x})j}$ denote the lateral distance between the winning neuron $i(\mathbf{x})$ and its neighbor j:

$$h_{ji(\mathbf{x})} = \exp(-\|\mathbf{r}_j - \mathbf{r}_{i(\mathbf{x})}\|^2/2\sigma^2) \qquad \textbf{(4.70)}$$

where $\mathbf{r}_j$ and $\mathbf{r}_{i(\mathbf{x})}$ are discrete vectors that indicate the coordinates of cells j and $i(\mathbf{x})$, respectively, on the bi-dimensional grid of the SOM. In this case, the value of $\sigma = \sigma(t)$ will be responsible for controlling the neighborhood influence along the iterative procedure of adaptation. N_{eigh} can be kept constant and $\sigma(t)$ iteratively reduced until the network weights stabilize.

Adaptation

The updating process for the weight vectors is similar to the one presented in Equation (4.19) for competitive learning schemes, but takes into account the topological neighborhood of the winning unit (Figure 4.40).

If the neighborhood is defined topologically as presented in Figure 4.38, then the updating rule presented in Equation (4.71) can be used to adjust the weight vectors.

$$\mathbf{w}_i(t+1) = \begin{cases} \mathbf{w}_i(t) + \alpha(t)[\mathbf{x}(t) - \mathbf{w}_i(t)] & \text{if } i \in N_{eigh} \\ \mathbf{w}_i(t) & \text{if } i \notin N_{eigh} \end{cases} \qquad \textbf{(4.71)}$$

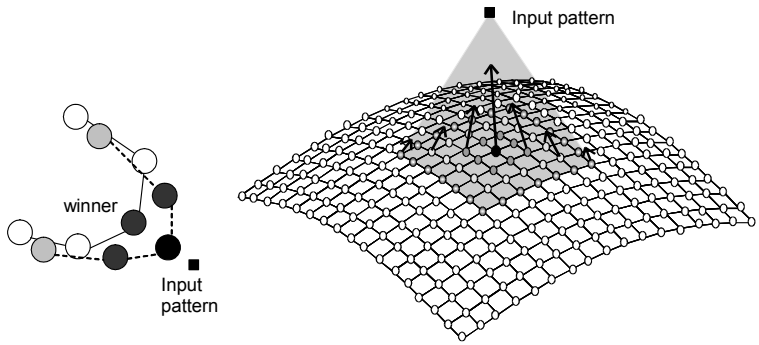


Figure 4.40: Adaptive procedure. The winner and its neighbors are moved in the direction of the input pattern. The winner unit performs a greater updating in the direction of the input pattern, followed by its immediate neighbors. (a) Uni-dimensional neighborhood. (b) Bi-dimensional neighborhood. (Courtesy of © Lalinka C. T. Gomes)

where the learning rate $\alpha(t)$ decreases with time, $0 < \alpha(t) < 1$.

However, if the neighborhood is defined using a general kernel function, an alternative notation is to introduce the scalar kernel function $h_{ji(\mathbf{x})} = h_{ji(\mathbf{x})}(t)$, described previously, directly in the weight adjusting equation

$$\mathbf{w}_i(t+1) = \mathbf{w}_i(t) + \alpha(t)\, h_{ji(\mathbf{x})}(t)\, [\mathbf{x}(t) - \mathbf{w}_i(t)] \qquad \textbf{(4.72)}$$

where $h_{ji(\mathbf{x})}(t) = 1$ within N_{eigh}, and $h_{ji(\mathbf{x})}(t) = 0$ outside N_{eigh}.

It is sometimes useful to keep the initial values for the parameters α and N_{eigh} (or σ) fixed for a number of iterations, usually 1,000, before starting to cool (reduce) them. Following this approach may result in an initial period corresponding to a self-organizing or *ordering phase* of the map, followed by a *convergence phase* in which, as the values of α and N_{eigh} (or σ) are monotonically reduced, the network will later suffer very little adaptation. Algorithm 4.5 summarizes the learning algorithm for the Kohonen self-organizing map. One simple form of cooling α and σ is by adopting a geometric decrease, i.e., by multiplying α and σ by a constant value smaller than one. Another form is by adopting the equations below (Haykin, 1999):

$$\alpha(t) = \alpha_0 \exp(-t/\tau_1) \qquad \textbf{(4.73)}$$

$$\sigma(t) = \sigma_0 \exp(-t/\tau_2) \qquad \textbf{(4.74)}$$

where $\alpha_0 \approx 0.1$, $\tau_1 \approx 1{,}000$, σ_0 = radius of the grid (all neurons are assumed to be neighbors of the winning neuron in the beginning of learning), and $\tau_2 = 1{,}000/\log(\sigma_0)$.

```
procedure [W] = som(max_it,α0,σ0,τ1,τ2,X)
    initialize W     //small random values (wi≠wj for i≠j)
    t ← 1
    while t < max_it do,
        vet_permut ← randperm(N)  //permutations of N
        for j from 1 to N do,      //for all input patterns
            //select the index i of pattern xi to be presented
            i ← vet_permut(j)  //present patterns randomly
            //competition and determination of the winner i(x)
            i(x) ← arg min_j ||xi – wj(t)||, j = 1,…,o
            //calculate the neighborhood function
            h_ji(x) = exp(–||rj – r_i(x)||^2/2σ^2)
            //cooperation and adaptation
            wi(t+1) = wi(t) + α(t) h_ji(x)(t) [x(t) – wi(t)]
        end  for
        //update α and the neighborhood function
        α ← reduce(α)        //until α reaches a minimal value
        σ ← reduce(σ)        //until only the winner is updated
        t ← t + 1
    end while
end procedure
```

Algorithm 4.5: Learning algorithm for the Kohonen self-organized map (SOM).

Biological Basis and Inspiration for the Self-Organizing Map

It has been long since a topographical organization of the brain, in particular of the cerebral cortex, has been suggested. These could be deduced from functional deficits and behavioral impairments produced by various kinds of lesions, hemorrhages, tumors, and malformations. Different regions in the brain thereby seemed to be dedicated to specific tasks (see Figure 4.5).

Nowadays, more advanced techniques, such as *positron emission topography* (PET), *magneto encephalography* (MEG), and *electroencephalogram* (EEG), can be used to measure nervous activity, thus allowing the study of the behavior of living brains. After a large number of experimentation and observation, a fairly detailed organization view of the brain has been proposed. Especially in higher animals, such as the human beings, the various cortices in the cell mass seem to contain many kinds of topographic maps, such that a particular location of the neural response in the map often directly corresponds to a specific modality and quality of sensory signal.

The development of the self-organizing maps as neural models is motivated by this distinctive feature of local brain organization into topologically ordered maps. The computational map constitutes a basic building block in the information-processing infrastructure of the nervous system. A computational map is defined by an array of neurons representing slightly different tuned processors or filters, which operate on the sensory information-bearing signals in parallel (Haykin, 1999).

The use of these computational maps offers some interesting properties:

- At each stage of representation, each incoming piece of information is kept in its proper context.
- Neurons dealing with closely related pieces of information are close together so they can interact via short synaptic connections.

The self-organizing map, or perhaps more accurately named *artificial topographic map*, was designed to be a system capable of learning through self-organization in a neurobiologically inspired manner. Under this perspective, the most relevant aspect of the nervous system to be taken into account is its capability of topographical map formation. As stated by T. Kohonen:

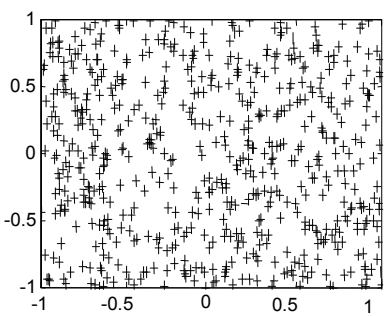
> "The spatial location or coordinates of a cell in the network then correspond to a particular domain of input signal patterns." (Kohonen, 1990; p. 1464)

Examples of Applications

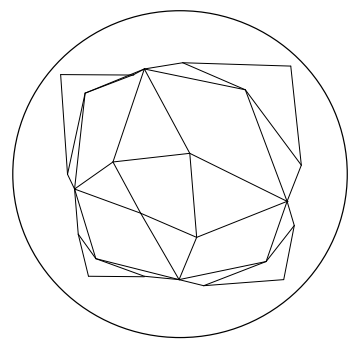
To illustrate the behavior and applicability of the SOM, it will first be applied to a structured random distribution of data. This will demonstrate the self-organizing properties of the network. The second problem - animals' data set - demonstrates the SOM capability of identifying groups of data, and allowing for the visualization of useful relationships within a high-dimensional data set in a low dimensional grid, usually of one or two dimensions.

Structured Random Distribution

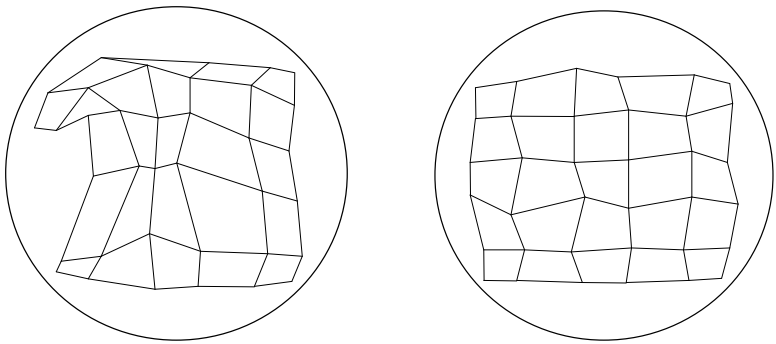
To illustrate the behavior of the SOM, consider an input data set composed of 500 bi-dimensional samples uniformly distributed over the interval [−1,1], as depicted in Figure 4.41(a). The input patterns were normalized to the unit interval so that their distribution lies on the surface of a unit circle. A bi-dimensional grid with six rows and six columns (6 × 6) of neurons is used. Figure 4.41(b) to (d) shows three stages of the adaptive process as the network learns to represent the input distribution.



(a)



(b)



(c) (d)

Figure 4.41: Application of the SOM network with a bi-dimensional grid to a randomly distributed input data set. (a) Input data set for a bi-dimensional SOM. (b) Condition of the grid after 1,000 iterations. (c) Condition of the grid after 5,000 iterations. (d) Grid after 10,000 iterations.

Semantic Maps

This section illustrates the SOM behavior by applying it to the Animals data set. This high-dimensional data set was originally proposed by Ritter and Kohonen (1989) to verify the SOM capability of creating a topographic map based on a symbolic data set. The data set is composed of 16 input vectors each representing an animal with the binary feature attributes as shown in Table 4.1.

Table 4.1: Animals data set: animals' names and their attributes.

		Dove	Hen	Duck	Goose	Owl	Hawk	Eagle	Fox	Dog	Wolf	Cat	Tiger	Lion	Horse	Zebra	Cow
Is	Small	1	1	1	1	1	1	0	0	0	0	1	0	0	0	0	0
	Medium	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0
	Big	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
Has	Two legs	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
	Four legs	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
	Hair	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
	Hooves	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
	Mane	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1	0
	Feathers	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
Likes to	Hunt	0	0	0	0	1	1	1	1	0	1	1	1	1	0	0	0
	Run	0	0	0	0	0	0	0	0	1	1	0	1	1	1	1	0
	Fly	1	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0
	Swim	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0

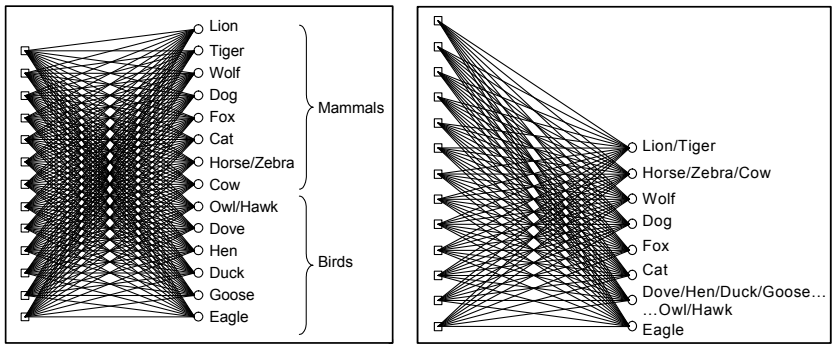


Figure 4.42: Uni-dimensional map containing labeled units with strongest responses to their respective inputs. (a) Output grid with 14 units. (b) Output grid with 8 units.

A value of '1' in Table 4.1 corresponds to the presence of an attribute, while a value of '0' corresponds to the lack of this attribute. The authors suggested that the interestingness of this data set lies in the fact that the relationship between the different symbols may not be directly detectable from their encodings, thus not presuming any metric relations even when the symbols represent similar items.

Figure 4.42 illustrates the performance of a SOM with a uni-dimensional output grid applied to the Animals data set. In Figure 4.42(a) an output grid with 14 units was used and in Figure 4.42(b) a grid with 8 units was used. Note the interesting capability of the SOM to cluster data which are semantically similar without using any labeled information about the input patterns. By observing Table 4.1 it is possible to note that the horse and zebra, as well as the owl and the hawk, have the same set of attributes. Note that animals with similar features are mapped onto the same or neighboring neurons.

4.4.6. Discrete Hopfield Network

J. Hopfield (1982, 1984) realized that in physical systems composed of a large number of simple elements, interactions among large numbers of elementary components yield complex collective phenomena, such as the stable magnetic orientations and domains in a magnetic system or the vortex patterns in fluid flow. Particularly, Hopfield noticed that there are classes of physical systems whose spontaneous behavior can be used as a form of general and error-correcting *content-addressable memory* (CAM). The primary function of a content-addressable memory, also called an *associative memory*, is to retrieve a pattern stored in the memory as a response to the presentation of an incomplete or noisy version of that pattern, as illustrated in Figure 4.43. In a CAM, the memory is reached not by knowing its address, but rather by supplying some subpart of the memory.

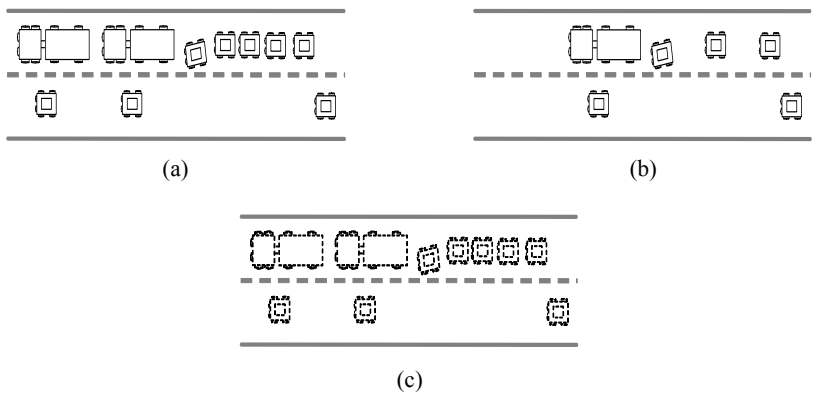


Figure 4.43: Content-addressable memory (CAM) or associative memory. Given a memorized pattern, the role of the CAM is to restore the stored pattern given an incomplete or noisy version of it. (a) Memorized patterns. (b) Incomplete patterns. (c) Noisy patterns.

By assuming that the time evolution of a physical system can be described by a set of general coordinates, then a point in a space, called state space, represents the instantaneous condition of the system. Therefore, the equations of motion of the system describe a flow in state space. Assuming also systems whose flow is performed toward locally stable points from anywhere within given neighborhoods of these stable points, these systems become particularly useful as content-addressable memories.

The original neuron model used by Hopfield (1982) was that of McCulloch and Pitts (1943), whose output could be either '0' or '1', and the inputs were external and from other neurons as well. Thus, his analysis involved networks with full back-coupling; that is, recurrent networks (such as the one illustrated in Figure 4.13, but with no external inputs). Other important features of the Hopfield network were the storage (instead of iterative learning) of the memories of the system, and the use of an asynchronous procedure of memory retrieval, as will be further discussed.

Recurrent Neural Networks as Nonlinear Dynamical Systems

The Hopfield neural networks were proposed by J. Hopfield in the early 1980s (Hopfield, 1982, 1984) as neural models derived from statistical physics. They incorporate a fundamental principle from physics: the storage of information in a dynamically stable configuration. Each pattern is stored in a valley of an energy surface. The nonlinear dynamics of the network is such that the energy is minimized and valleys represent points of stable equilibrium, each one with its own basin of attraction. Therefore, a network memory corresponds to a point of stable equilibrium, thus allowing the system to behave as an associative memory. Appendix B.4.3 brings an elementary discussion about nonlinear dynamical systems.

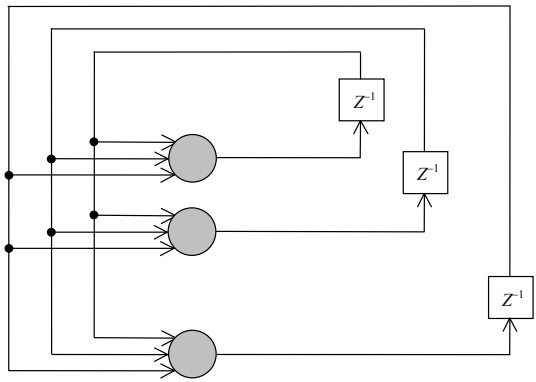


Figure 4.44: Recurrent neural network with no hidden layer.

In this case, convergent flow to stable states becomes the essential feature of the CAM operation. There is a simple mathematical condition that guarantees that the state space flow algorithm converges to stable states. Any symmetric matrix **W** with zero-diagonal elements (i.e., $w_{ij} = w_{ji}$, $w_{ii} = 0$) will produce such a flow. The proof of this property follows from the construction of an appropriate energy function that is always decreased by any state change produced by the algorithm.

Assume a single-layer neural network composed of N neurons and a corresponding set of unit delays, forming a multiple-loop feedback system, as illustrated Figure 4.44. Note that this network is a particular case of the network presented in Figure 4.13.

Assume that the neurons in this network have symmetric coupling described by $w_{ij} = w_{ji}$ ($i,j = 1, \ldots, N$), where w_{ij} is the synaptic strength connecting the output of unit i to the input of unit j. Assume also that no self-feedback is allowed, i.e., $w_{ii} = 0$ ($i = 1, \ldots, N$). Let $u_j(t)$ be the net input to neuron j and $y_j(t)$ its corresponding output $y_j(t) = f(u_j(t))$, where $f(\cdot)$ is the activation function of unit j.

Let now the vector $\mathbf{u}(t)$ be the state of the network at time t. Then, from the basic integrate and fire model of a neuron given by Equation (4.1) it is possible to define an additive model of a neuron as given by Equation (4.75).

$$\tau_m \frac{du(t)}{dt} = u_{res} - u(t) + R_m i(t) \qquad \textbf{(4.1)}$$

where τ_m is the membrane time constant of the neuron determined by the average conductance of the channels (among other things); u_{res} is the resting potential of the neuron; $i(t)$ is the input current given by the sum of the synaptic currents generated by firings of presynaptic neurons; and R_m is the resistance of the neuron to the flow of current (ions).

$$C_j \frac{du_j}{dt} = I_j - \frac{u_j}{R_j} + \sum_{\substack{i=1 \\ i \neq j}}^{N} w_{ji} f_i(u_i), j = 1, \ldots, N \quad \textbf{(4.75)}$$

where R_j and C_j are the leakage resistance and capacitance of the neuron respectively, I_j corresponds to an externally applied bias, u_j is the net input to unit j, and $f_i(\cdot)$ is the activation function of unit i.

To the dynamics provided by Equation (4.75) it is possible to derive an energy function as given by Equation (4.76) (for details see Hopfield, 1984).

$$E = -\frac{1}{2}\sum_{i=1}^{N}\sum_{\substack{j=1 \\ j \neq i}}^{N} w_{ji} y_i y_j + \sum_{j=1}^{N} \frac{1}{R_j} \int_0^{y_j} f_j^{-1}(z)dz + \sum_{j=1}^{N} b_j y_j \quad \textbf{(4.76)}$$

This energy function is a monotonically decreasing function of the state of the network. Therefore, the stationary states of this network are such that they correspond to equilibrium points (Hopfield, 1984). These equilibrium points are termed attractors because there is a neighborhood (basin of attraction) over which these points exert a dominant influence.

Discrete Hopfield Network

The discrete Hopfield networks constitute a particular case of recurrent networks for which the state space is discrete and the neuron models are the ones proposed by McCulloch and Pitts (Section 4.3.1). This allows it to be seen as a content-addressable memory or nonlinear associative memory. In this case, the equilibrium points of the network are known to correspond to the patterns to be memorized. However, the network weights that produce the desired equilibrium points are not known, and thus must be determined according to some rule.

Suppose the goal is to memorize a set $\mathbf{X}$ of m-dimensional vectors (bipolar strings built from the set $\{-1,1\}$), denoted by $\{\mathbf{x}_i \mid i = 1, \ldots, N\}$ in a network with m neurons. These vectors are called the *fundamental memories* of the system and represent the patterns to be memorized. According to the generalized Hebb's rule, the weight matrix $\mathbf{W} \in \Re^{m \times m}$ of the network is determined by, in vector notation:

$$\mathbf{W} = \frac{1}{m}\sum_{i=1}^{N} \mathbf{x}_i (\mathbf{x}_i)^T \quad \textbf{(4.77)}$$

As discussed previously, to guarantee the convergence of the algorithm to stable states, it is necessary to set $w_{ii} = 0$. Therefore, Equation (4.77) becomes

$$\mathbf{W} = \frac{1}{m}\sum_{i=1}^{N} \mathbf{x}_i (\mathbf{x}_i)^T - \frac{N}{m}\mathbf{I} \quad \textbf{(4.78)}$$

where $\mathbf{I}$ is the identity matrix.

Equation (4.78) satisfies all the conditions required for the convergence of this discrete network to stable states: it is symmetric and with zero-valued diagonal. A monotonically decreasing energy function can thus be defined (see Hopfield 1982).

```
procedure [W,Y] = hopfield(X,Z)
    //storage phase
    W ← (1/m)X.X^T – (N/m)I
    //retrieval phase
    vet_permut ← randperm(m)        //permutations of n
    for j from 1 to m do,           //for all input patterns
        i ← vet_permut(j)           //randomly select a neuron
        while y_i is changing do,
            y_i ← sign(W.z_i)
        end while
    end for
end procedure
```

Algorithm 4.6: Discrete Hopfield network.

After storing the memories in the network (*storage phase*), its retrieval capability can now be assessed by presenting new or noisy patterns to the network (*retrieval phase*). Assume an m-dimensional vector $\mathbf{z} \in \Re^{m\times 1}$ is taken and presented to the network as its initial state. The vector $\mathbf{z}$, called *probe*, can be an incomplete or noisy version of one of the fundamental memories of the network.

The memory retrieval process obeys a dynamical rule termed *asynchronous updating*. Randomly select a neuron j of the network as follows:

$$s_j(0) = z_j;\ \forall j$$

where $s_j(0)$ is the state of neuron j at iteration $t = 0$, and z_j is the j-th element of vector $\mathbf{z}$. At each iteration, update one randomly selected element of the state vector $\mathbf{s}$ according to the rule:

$$s_j(t+1) = \text{sign}\left[\sum_{i=1}^{m} w_{ji} s_i(t)\right],\ j = 1, \ldots, m \qquad \textbf{(4.79)}$$

where the function sign(u_j) is the signum function: it returns 1 if $u_j > 0$, -1 if $u_j < 0$, and remains unchanged if $u_j = 0$.

Repeat the iteration until the state vector $\mathbf{s}$ remains unchanged. The fixed point, stable state, computed ($\mathbf{s}_{\text{fixed}}$) corresponds to the output vector returned by the network: $\mathbf{y} = \mathbf{s}_{\text{fixed}}$.

Let $\mathbf{X} \in \Re^{m\times N}$ be the matrix of N input patterns of dimension m each (the patterns are placed columnwise in matrix $\mathbf{X}$), and $\mathbf{Z} \in \Re^{m\times n}$ be the matrix of n probe patterns of dimension m each. Algorithm 4.6 presents the pseudocode for the discrete Hopfield network, where $\mathbf{Y} \in \Re^{m\times N}$ is the matrix of restored patterns.

Spurious Attractors

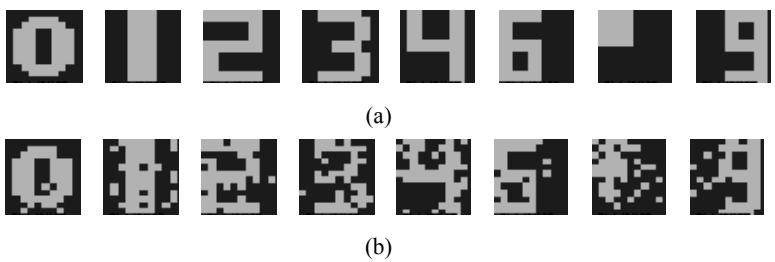
If the Hopfield network stores N fundamental memories using the generalized Hebb's rule, the stable states present in the energy surface are not restricted to the fundamental memories stored. Any stable state that is not associated with the fundamental memories are named *spurious attractors*. Spurious attractors are

due to several factors. The energy function is symmetrical; any linear combination of an odd number of stable states is also a stable state; and for a large number of fundamental memories the energy function generates stable states that are not correlated with any of the fundamental memories.

Example of Application

To illustrate the error-correcting capability of the Hopfield network, consider the same character recognition problem discussed in Section 4.4.2 and illustrated in Figure 4.45(a). In the present case, the patterns are assumed to be bipolar, that is, composed of a bit −1 (dark gray) or +1 (light gray). As each pattern has a dimension $m = 120$, the network used in the experiment has $m = 120$ neurons, and therefore $m^2 - m = 12{,}280$ connections.

To test the storage capability of the network, the same patterns (fundamental memories) can be presented to the network. To demonstrate its error-correcting capability, introduce random noise in the patterns by independently reversing some pixels from +1 to −1, or vice-versa, with a given probability. The corrupted (noisy) patterns are used as probes to the network (Figure 4.45(b)).



(a)

(b)

Figure 4.45: (a) Fundamental memories for the Hopfield network. (b) Noisy patterns.

4.5 FROM NATURAL TO ARTIFICIAL NEURAL NETWORKS

Table 4.2 summarizes the interpretation of biological neural networks as artificial neural networks. The study of any mathematical model involves the mapping of the mathematical domain into some observable part of the natural world. The nervous system studied here includes neurons, synaptic strengths, neural firings, networks of neurons, and several other features of the brain viewed at a specific level of description. In Section 4.4, where the typical neural networks were described, the models introduced were invariably placed in the biological context. Even in the case of the Hopfield network, where this description was not so explicit, it was discussed that a continuous model of such network uses a simple integrate-and-fire neuron, and thus the network can also be viewed as searching for minima in an energy surface.

Table 4.2: Interpretation of the biological terminology into the computational domain.

Biology (Nervous System)	Artificial Neural Networks
Neuron	Unit, node, or (artificial) neuron
Synaptic contact	Connection
Synaptic efficacy (such as the number of vesicles released with each action potential, the amount of neurotransmitter within each vesicle, and the total number of channel receptors exposed to neurotransmitters)	Weight, connection strength, or synaptic strength
Neural integration over the input signals	Net input to the neuron
Triggering of an action potential (spike)	Output of the neuron given by an activation function applied to its net input
Limited range firing rate	Limited output of the processing strength by an activation function
Threshold of activation	Bias
Laminae (layers) of neurons	Layers of neurons
Network of neurons	Network of neurons
Excitation/inhibition	Positive/negative weight

4.6 SCOPE OF NEUROCOMPUTING

Artificial neural networks can, in principle, compute any computable function, i.e., they can do everything a standard digital computer can do, and can thus be classified as a type of *neurocomputer*. Artificial neural networks are particularly useful for clustering, classification, and function approximation (mapping) problems to which hard and fast rules cannot be easily applied. As discussed in Section 4.4.4, multi-layer perceptrons are universal function approximators, thus any finite-dimensional vector function on a compact set can be approximated to an arbitrary precision by these networks. Self-organizing feature maps, like the one described in this chapter, find applications in diverse fields, such as unsupervised categorization of data (clustering), pattern recognition, remote sensing, robotics, processing of semantic information, and knowledge discovery in databases (data mining). Recurrent networks have been broadly applied to time series prediction, pattern recognition, system identification and control, and natural language processing. There are also many other types of neural networks and learning algorithms which find applications in different domains.

Vasts lists of practical and commercial applications of ANN, including links to other sources, can be found at the Frequently Asked Questions (FAQ) site for neural networks:

ftp://ftp.sas.com/pub/neural/FAQ7.html#A_applications.

Similarly to the case of evolutionary algorithms, the applications range from industry to arts to games.

4.7 SUMMARY

The nervous system is one of the major control systems of living organisms. It is responsible, among other things, for receiving stimuli from the environment, processing them, and then promoting an output response. It has been very influential in the design of computational tools for problem solving that use neuron-like elements as the main processing units.

In a simplified form, the nervous system is composed of nerve cells, mainly neurons, structured in networks of neurons. Through synaptic modulation the nervous system is capable of generating internal representations of environmental stimuli, to the point that these stimuli become unnecessary for the emergence of ideas and creativity. The internal models themselves are capable of self-stimulation.

Artificial neural networks were designed as highly abstract models based upon simple mathematical models of biological neurons. The assembly of artificial neurons in network structures allows the creation of powerful neurocomputing devices, with applications to several domains, from pattern recognition to control.

This chapter only scratched the surface of the mature and vast field of neural networks. It presented a framework to design ANN (neuron models, network structures, and learning algorithms), and then proceeded with a review of the standard most well-known types of neural networks developed with inspiration in the nervous systems: Hebb net, Perceptron network, ADALINE, MLP, SOM, and discrete Hopfield network. There are several other types of neural networks that receive great attention and have applications in various domains, but that we could not talk about here due to space constraints. Among these, it is possible to remark the *support vector machines* (Cortes and Vapnik, 1995; Vapnik, 1998) and the *radial basis function* (Broomhead and Lowe, 1988; Freeman and Saad, 1995) neural networks. Appendix C provides a quick guide to the literature with particular emphasis on textbooks dedicated to neural networks. All readers interested in knowing more about the many types of neural networks may consult the literature cited.

4.8 EXERCISES

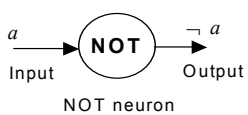
4.8.1. Questions

1. It has been discussed, in Section 4.2, that a neuron can fire by receiving input signals from other neurons. In most cases it is assumed that these signals come from the environment through sensory cells in the skin, eyes, and so forth. However, it is also known that the brain is capable of presenting thought processes from stimuli generated within the brain itself. Discuss how these “self-stimuli” can be generated.

2. Artificial neural networks have been extensively used in pattern recognition tasks, such as character and face recognition. Independently of the potentiality of the network implemented, would you expect an ANN to outperform human beings, for instance in noisy character recognition? Justify your answer.
3. Classify the list of topics below as fact or fiction (argument in favor of your classification and cite works from the literature as support):
    a) We only use 10% of the potential of our brains.
    b) The number of neurons we are born with is the number of neurons we die with.
    c) Neurons cannot reproduce.
    d) Men have more neurons than women.
    e) Women have a broader peripherical vision than men.

4.8.2. Computational Exercises

1. Section 4.3.1 introduced the McCulloch and Pitts neuron and presented one form of implementing it so as to simulate the logical AND and OR functions. Assuming there is a NOT neuron, as illustrated in Figure 4.46, connect some AND, OR, and NOT neurons such that the resulting artificial neural network performs the XOR function with inputs *a* and *b*. Draw the resultant network and describe the required threshold for each neuron in the network.



a	*b*	*a* XOR *b*
0	0	0
0	1	1
1	0	1
1	1	0

Figure 4.46: A NOT neuron and the truth-table for the XOR function.

2. In Section 4.4.1, the extended Hebbian updating rule for the weight vector was presented. Write a pseudocode for the Hebb network assuming that the network is composed of a single neuron with linear activation and is going to be used for supervised learning:

    Input patterns: $\mathbf{X} \in \Re^{m \times N}$.

    Desired outputs: $\mathbf{d} \in \Re^{1 \times N}$.

    Network outputs: $\mathbf{y} \in \Re^{1 \times N}$.

    The procedure should return as outputs the network output y for each training pattern, the weight vector $\mathbf{w}$ and the bias term b.

Inputs		Desired output
x_1	x_2	x_1 AND x_2
0	0	0
0	1	0
1	0	0
1	1	1

Inputs		Desired output
x_1	x_2	x_1 OR x_2
0	0	0
0	1	1
1	0	1
1	1	1

Figure 4.47: Boolean functions AND and OR with binary outputs.

Inputs		Desired output
x_1	x_2	x_1 AND x_2
0	0	–1
0	1	–1
1	0	–1
1	1	+1

Inputs		Desired output
x_1	x_2	x_1 OR x_2
0	0	–1
0	1	+1
1	0	+1
1	1	+1

Figure 4.48: Boolean functions AND and OR with bipolar outputs.

3. Apply the extended Hebb network procedure, written in the previous exercise to realize the Boolean functions AND and OR, as illustrated in Figure 4.47.

   What happens when the first, second, and third patterns are presented to the network in the AND function? The same happens with the first pattern of function OR. Explain why the learning rule is not capable of generating a correct decision line?

   Change the binary vector of desired outputs by a bipolar vector, that is, every value '0' in vector **d** should correspond to a value '–1', as illustrated in Figure 4.48, and reapply the Hebb network to this problem. What happens? Is the network capable of solving the problem? Justify your answer.

   Finally, change the input patterns into the bipolar form and try to solve the problem using the simple Hebb network.

4. Apply the simple perceptron algorithm (Algorithm 4.1) to the AND and OR functions with binary inputs and desired outputs. Initialize the weight vector and bias all with zero, and with small random numbers as well. Compare the results of the different initialization methods and draw the decision lines.

5. Apply the simple perceptron algorithm (Algorithm 4.1) to the XOR function. Is it capable of finding a decision line to correctly classify the inputs into '1' or '0'? Why using bipolar inputs or desired outputs still does not solve the problem in this case?

6. Implement the perceptron algorithm for multiple output neurons (Algorithm 4.2) and train the network to classify the characters presented in Figure 4.23.

   Using the weight matrix and bias vector trained with the noise-free characters, insert some random noise in these patterns and test the network sensitivity in relation to the noise level; that is, test how the network perfor-

mance degrades in relation to the level of noise introduced in the test set. Try various levels of noise, from 5% to 50%.

7. Determine the gradient of an arbitrary bias for an ADALINE network with linear output nodes. That is,

   Determine $\frac{\partial \Im}{\partial b_I}$, where $\Im = \sum_{i=1}^{o} e_i^2 = \sum_{i=1}^{o} (d_i - y_i)^2$, and $y_i = \Sigma_j w_{ij} x_j + b_i$, $j = 1, \ldots, m$.

8. Write a pseudocode for the ADALINE network and compare it with the pseudocode for the Perceptron network.

9. Repeat Exercise 7 above using an Adaline network. Compare the results with those obtained using the perceptron.

10. Algorithm 4.3 presents the algorithm to run the single-layer perceptron network. Write a pseudocode for an algorithm to run the multi-layer perceptron (MLP) network. Assume generic activation functions and a number of hidden layers to be entered by the user. (The trained weight matrices and bias vectors must be entered as inputs to the algorithm.)

11. Apply an MLP network with two sigmoid hidden units and a single linear output unit trained with the standard backpropagation learning algorithm to solve the XOR problem.

    Test the network sensitivity to various values of α; that is, test the convergence time of the algorithm for different values of α. Tip: start with $\alpha = 0.001$ and vary it until $\alpha = 0.5$.

    For a fixed value of α draw the decision boundaries constructed for each hidden unit, and the decision boundary constructed by the complete network.

    Can you solve this problem with a single hidden unit? Justify your answer.

12. Apply an MLP network with threshold (McCulloch and Pitts) units, two hidden and a single output unit, trained with the backpropagation learning algorithm to solve the XOR problem.

    Determine the weights of the network and draw the decision boundaries constructed for each hidden unit, and the decision boundary constructed by the complete network.

13. Repeat the previous exercise assuming the neural network has sigmoidal activation functions in the hidden and output layers.

14. It is known that the network presented in Figure 4.49 is capable of solving the XOR problem with a single hidden unit.

    Can you apply the backpropagation algorithm to train this network? If yes, how? If not, why?

    Find an appropriate set of weights so that the network with sigmoid hidden units and linear (or sigmoid) output unit solves the XOR problem.

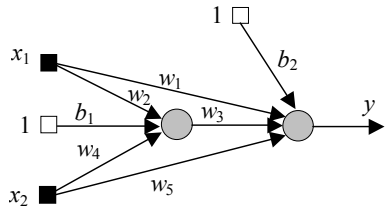


Figure 4.49: Network with connections from the input units directly to the output unit.

15. Train an MLP network to approximate one period of the function $sin(x).cos(2x)$, $x \in [0,2\pi]$, with only 36 training samples uniformly distributed over the domain of x. Test the network sensitivity with relation to:
    a) The number of hidden units.
    b) The learning rate α.
16. Insert some noise into the data set of the previous exercise and retrain the MLP network with this new data set. Study the generalization capability of this network (Section 4.4.4).
17. Apply the MLP network trained with the backpropagation learning algorithm to solve the character recognition task of Section 4.4.2.

    Determine a suitable network architecture and then test the resultant network sensitivity to noise in the test data. Test different noise levels, from 5% to 50%.
18. Apply an MLP network to solve the two-spirals problems (Figure 4.50) using in the previous chapter to illustrate the application of genetic programming. This data set is composed of 190 labeled samples (95 samples in each class), generated using polar coordinates.

    Determine a suitable number of neurons in the MLP network to be trained using the standard backpropagation algorithm. Assume the algorithm has converged when $MSE \leq 10^{-3}$, and plot the decision surface generated separating both classes.

    Test the sensitivity of the trained network to various levels of noise added to the input data.
19. The *encoding/decoding* (ENC/DEC) problem is a sort of *parity* problem; that is, the output is required to be one if the input pattern contains an odd number of ones, and zero otherwise. Assuming a data set with 10 samples ($N = 10$) of dimension 10 each ($m = 10$), and a network with 10 outputs ($o = 10$), the input vector is composed of N bits, from which only one has a value of '1', and the others have values '0'. The desired output vector is identical to the input vector so that the network has to make an auto-association between the input and output values. The goal is to learn a map-

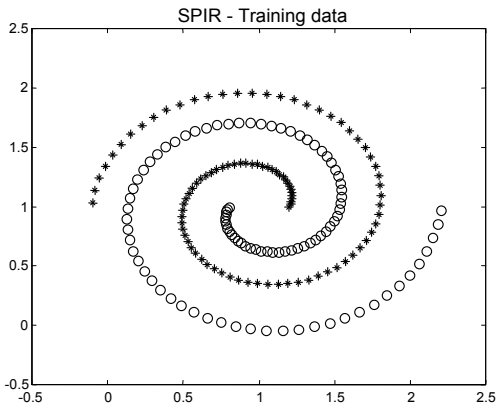


Figure 4.50: Input data set for training a classifier.

ping from the m inputs to the h hidden units (*encoding*) and then from the h hidden units to the o outputs (*decoding*). Usually, in a network m-h-o (m inputs, h hidden units, and o outputs) $h < N$, where N is the number of input patterns. If $h \leq \log_2 N$ it is said to be a tight encoder.

Train an MLP network with $h = 10$ to solve an ENC/DEC problem of dimension 10.

Is it possible to train an MLP network so that it becomes a tight encoder?

20. Apply the self-organizing map (SOM) learning algorithm to the clustering problem presented in Section 5.6.2, Exercise 3 of the Swarm Intelligence chapter (Chapter 5). Try first a uni-dimensional output grid, and then a bi-dimensional output grid. Compare and discuss the results of each method.
21. Apply the self-organized map to the Animals data set presented in Section 4.4.5. Consider a bi-dimensional output grid with 10 rows by 10 columns of neurons (10×10).
22. Apply a bi-dimensional SOM to the two-spirals problem considering unlabeled data. Depict (plot) the final configuration of the neurons in the network.
23. Implement the discrete Hopfield network and apply it to the character recognition problem, as described in Section 4.4.2. Test the network sensitivity to noise, that is, introduce noise in the patterns with increasing probability (5%, 10%, 15%, 20%, 25%, … , 50%) and study the error-correcting capability of the network.
24. A more efficient rule, than the generalized Hebb rule, to define the weight matrix of a discrete Hopfield network is the *projection rule* or *pseudo-inverse rule*. Although this rule does not present the biological motivation of the Hebb rule, it explores some algebraic features of the equilibrium points.

A vector $\mathbf{x}_i$ is going to be an equilibrium point if $\mathbf{W}\mathbf{x}_i = \mathbf{x}_i$, $i = 1, \ldots, n$. Let $\mathbf{X}$ be the matrix of all input patterns, then $\mathbf{W}\,\mathbf{X} = \mathbf{X}$. A solution to this equation can be given by the following expression:

$$\mathbf{W} = \mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$$

The necessary condition for $(\mathbf{X}^T\mathbf{X})^{-1}$ to exist is that the vectors $\mathbf{x}_i$, $i = 1, \ldots, n$ be linearly independent, and thus, matrix $\mathbf{X}$ has full rank.

Test if the input patterns used in the previous exercise are linearly independent (see Appendix B.1.2) and use the projection rule to store the patterns in the discrete Hopfield network.

Test the error-correcting capability of the network trained with this rule and compare with the results obtained in the previous exercise.

4.8.3. Thought Exercises

1. Discuss the similarities and differences between the perceptron and the ADALINE updating rules. Why is the ADALINE more powerful than the perceptron algorithm?
2. In Appendix B.4.4 some elements of graph theory are introduced. Provide a graphical description of a neural network, more specifically, propose a signal-flow graph of the artificial neuron depicted in Figure 4.9.
3. Chapter 3 introduced the concept of a fitness or adaptive landscape whose contours represent different levels of adaptation to the environment. Populations of individuals at high levels (peaks) of this landscape are more adapted to the environment, while populations at lower levels (valleys) are less adapted. Compare the concept of a fitness landscape to the error surface discussed in Section 4.3.3.

4.8.4. Projects and Challenges

1. In Section 4.3.1 a basic integrate-and-fire neuron model was presented according to Equation (4.1) and Equation (4.3). Equation (4.1) corresponds to an inhomogeneous linear differential equation that can be solved numerically, and also analytically in some cases. Assuming that the sum of the input currents to the neuron $i(t)$ is constant; that is, $i(t) = i$, determine the analytical solution to Equation (4.1).

   Assume now that the activation threshold is set to $\theta = 10$. Plot a graph with the variation of the solution to Equation (4.1), $u(t)$, as a function of time t for $Ri = 8$.

   Finally, plot a graph with the variation of the solution to Equation (4.1), $u(t)$, as a function of time t for $Ri = 10$. If the neuron fires a spike, reset its membrane potential to a fixed value $u_{\text{res}} = 1$.

   Can you observe spikes in your graphs? If not, explain why.
2. In Section 4.4.5 the SOM learning algorithm was presented. A competitive learning rule based upon the minimization of the Euclidean distance bet-

ween the input pattern and the neurons' weights was presented. Another criterion to determine the winning neuron, i.e., the best matching neuron, is based on the maximization of the inner product between the weight vectors and the input pattern ($\mathbf{w}_j^T\mathbf{x}$).

Show that minimizing the Euclidean distance between a given input pattern and the neurons' weights is mathematically equivalent to maximizing the inner product $\mathbf{w}_j^T\mathbf{x}$.

Implement this new competitive rule and compare qualitatively the results with those obtained with the Euclidean distance.

Use the same initialization values for the network weights and the same parameters for the algorithm and compare quantitatively and qualitatively both competitive strategies.

3. Explain why a multi-layer feedforward neural network with sigmoidal output units cannot learn the simple function $y = 1/x$, with $x \in (0,1)$. How can you modify this network so that it can learn this function.

4. If an evolutionary algorithm is a general-purpose search and optimization technique, it can be applied to determine suitable network architectures and weight values for a neural network. Implement an evolutionary algorithm to determine a suitable MLP network architecture trained with the backpropagation algorithm to solve the IRIS problem described below.

   There are a vast number of data sets available for testing and benchmarking neural networks and learning algorithms. For instance, the University of California Irvine (UCI) provides many collections of data sets accessible via anonymous FTP at ftp.ics.uci.edu or their website at http://www.ics.uci.edu/~mlearn/MLRepository.html.

   The IRIS data set is a very well-known classification problem containing 150 samples divided into three differences classes, each of which with 50 patterns. The samples have a dimension of 4 ($m = 4$) and refer to a type of iris of plants. One of the three classes is linearly separable from the others, and two of them are not. Further documentation about this problem can be downloaded from the UCI repository together with the data set.

   Tip: define a suitable representation for the chromosomes, a fitness function, and appropriate genetic operators. Each chromosome will correspond to a neural network that will have to be trained via backpropagation in order to be evaluated. The objective is to find a network with a classification error less than 5%.

4.9 REFERENCES

[1] Anderson, J. A. (1995), *An Introduction to Neural Networks*, The MIT Press.

[2] Broomhead, D. S. and Lowe D. (1988), "Multivariable Functional Interpolation and Adaptive Networks", *Complex Systems*, **2**, pp. 321–355.

[3] Churchland, P. and Sejnowski, T. J. (1992), *The Computational Brain*, MIT Press.