Neural Networks

Rolf Pfeifer
Dana Damian
Rudolf Füchslin

Contents

CHAPTER 1

Introduction and motivation

The brain performs astonishing tasks: we can walk, talk, read, write, recognize hundreds of faces and objects, irrespective of distance, orientation and lighting conditions, we can drink from a cup, give a lecture, drive a car, do sports, we can take a course in neural networks at the university, and so on. Well, it's actually not the brain, it's entire humans that execute the tasks. The brain plays, of course, an essential role in this process, but it should be noted that the body itself, the morphology (the shape or the anatomy, the sensors, their position on the body), and the materials from which it is constructed also do a lot of useful work in intelligent behavior. The keyword here is embodiment, which is described in detail in [**Pfeifer and Scheier, 1999**] and [**Pfeifer and Bongard, 2007**]. In other words, the brain is always embedded into a physical system that interacts with the real world, and if we want to understand the function of the brain we must take embodiment into account.

Because the brain is so awesomly powerful, it seems natural to seek inspiration from the brain. In the field of neural computation (or neuro-informatics), the brain is viewed as performing computation and one tries to reproduce at least partially some of its amazing abilities. This type of computation is also called "brain-style" computing.

One of the well-known interesting characteristics of brains is that the behavior of the individual neuron is clearly not considered "intelligent" whereas the behavior of the brain as a whole is (again, we should also include the body in this argument). The technical term used here is *emergence*: if we are to understand the brain, we must understand how the global behavior of the brain-body system emerges from the activity and especially the interaction of many individual units.

In this course, we focus on the brain and the neural systems and we try to make proper abstractions so that we can not only improve our understanding of how natural brains function, but exploit brain-style computing for technological purposes. The term *neural networks* is often used as an umbrella term for all these activities.

Before digging into how this could actually be done, let us look at some areas where this kind of technology could be applied. In factory automation, number crunching, abstract symbol manipulation, or logical reasoning, it would not make sense because the standard methods of computing and control work extremely well, whereas for more "natural" forms of behavior, such as perception, movement, locomotion, and object manipulation, we can expect interesting results. There are a number of additional impressive characteristics of brains that we will now look at anecdotally, i.e. it is not a systematic collection but again illustrates the power of natural brains. For all of these capacities, traditional computing has to date not come up with generally accepted solutions.

1. Some differences between computers and brains

Here are a few examples of differences between biological brains and computers. The point of this comparison is to show that we might indeed benefit by employing brain-style computation.

Parallelism. Computers function, in essence, in a sequential manner, whereas brains are massively parallel. Moreover, the individual neurons are densely connected to other neurons: a neuron has between just a few and 10,000 connections. Of particular interest is the observation that parallelism requires learning or some other developmental processes. In most cases it is not possible to either set the parameters (the weights, see later) of the network manually, or to derive them in a straightforward way by means of a formula: a learning process is required. The human brain has roughly 10^{11} neurons and 10^{14} synapses, whereas modern computers, even parallel supercomputers, typically – with some exceptions – have no more than 1000 parallel processors. In addition, the individual "processing units" in the brain are relatively "simple" and very slow, whereas the processing units of computers are extremely sophisticated and fast (cycle times in the range of nanoseconds).

This point is illustrated by the *"100 step constraint"*. If a subject in a reaction time task is asked to press a button as soon as he or she has recognized a letter, say "A", this lasts roughly 1/2s. If we assume that the "operating cycle of a cognitive operation is on the order of 5-10ms, this yields a maximum of 200 operations per second. How is it possible that recognition can be achieved with only 200 cycles? The massive parallelism and the high connectivity of neural systems, as well as the fact that a lot of processing is performed right at the periphery (e.g. the retina), appear to be core factors.

Graceful degradation is a property of natural systems that modern computers lack to a large extent unless it is explicitly provided for. The term is used to designate systems that still operate - at least partially - if certain parts malfunction or if the situation changes in unexpected ways. Systems that display this property are

(a) noise and fault tolerant, and
(b) they can generalize.

Noise tolerance means that if there is noise in the data or inside the system the function is not impaired, at least not significantly. The same holds for fault tolerance: If certain parts malfunction, the system does not grind to a halt, but continues to work - depending on the amount of damage, of course. The ability to generalize means that if there is a situation the system has never encountered before, the system can function appropriately based on its experience with similar situations. Generalization implies that similar inputs lead to similar outputs. In this sense, the parity function (of which the XOR is an instance) does not generalize (if you change only 1 bit at the input, you get maximum change at the out; see chapter 2). This point is especially important whenever we are dealing with the real world because there no two situations are ever identical. This implies that if we are to function in the real world, we must be able to generalize.

Adaptivity/Learning: Another difference between computers and brains concerns their ability to learn. In fact, most natural systems learn continuously, as soon as there is a change in the environment. For humans it is impossible *not* to learn: once you are finished reading this sentence you will be changed forever, whether you like it or not. And learning is, of course, an essential characteristic of

any intelligent system. There is a large literature on learning systems, traditional and with neural networks. Neural networks are particularly interesting learning systems because they are massively parallel and distributed. Along with the ability to learn goes the ability to forget. Natural systems do forget whereas computer don't. Forgetting can, in a number of respects, be beneficial for the functioning of the organism: avoiding overload and unnecessary detail, generalization, forgetting undesirable experiences, focus on recent experiences, rather than on old ones, etc.

Learning always goes together with memory. The organization of memory in a computer is completely different from the one in the brain. Computer memories are accessed via addresses, there is a separation of program and data, and items once stored, are never forgotten, unless they are overwritten for some reason. Brains, by contrast, do not have "addresses", there is no separation of "programs" and "data", and, as mentioned above, they have a tendency to forget. When natural brains search for memories, they use an organizational principle which is called "associative memory" or "content-addressable memory": memories are accessed via part of the information searched for, not through an address. When asked what you had for lunch yesterday, you solve this problem by retrieving, for example, which class you attended before lunch, to which cafeteria you went etc., not by accessing a memory at a particular address (it is not even clear what an "address" in the brain would mean).

Also, computers don't "get tired", they function indefinitely, which is one of the important reasons why they are so incredibly useful. Natural brains get tired, they need to recover, and they occasionally need some sleep, learning.

Nonlinearity: Neurons are highly nonlinear, which is important particularly if the underlying physical mechanism responsible for the generation of the input signal is inherently (e.g. speech signal) nonlinear.

Recently, in many sciences, there has been an increasing interest in non-linear phenomena. If a system – an animal, a human, or a robot – is to cope with non-linearities, non-linear capacities are required. Many examples of such phenomena will be given throughout the class.

Plasticity: Learning and adaptivity are enabled by the enormous neural plasticity which is illustrated e.g. by the experiment of Melchner [**von Melchner et al., 2000**], in which the optic nerves of the eyes of a ferret were connected to the auditory cortex which then developed structures similar to the visual one.

The *Paradox of the expert* provides another illustration of the difference between brains and computers. At a somewhat anecdotal level, the paradox of the expert is an intriguing phenomenon which has captured the attention of psychologists and computer scientists alike. Traditional thinking suggests: the larger the database, i.e. the more comprehensive an individual's knowledge, the longer it takes to retrieve one particular item. This is certainly the case for database systems and knowledge-based systems no matter how clever the access mechanism. In human experts, the precise opposite seems to be the case: the more someone knows, the faster he or she can actually reproduce the required information. The parallelism and the high connectivity of natural neural systems are important factors underlying this amazing feat.

Context effects and constraint satisfaction. Naturally intelligent systems all have the ability to take context into account. This is illustrated in Figure 1 where

FIGURE 1. "tAe cHt". The center symbol in both words is identical but, because of context, is read as an H in the first case and as an A in the second.

the center letter is identical for both words, but we naturally, without much reflection, identify the one in the first word as an "H", and the one in the second word as an "A". The adjacent letters, which in this case form the context, provide the necessary constraints on the kinds of letters that are most likely to appear in this context. In understanding everyday natural language, context is also essential: if we understand the social situation in which an utterance is made, it is much easier to understand it, than out of context.

We could continue this list almost indefinitely. Because of the many favorable properties of natural brains, researchers in the field of neural networks have tried to harness some of them for the development of algorithms.

2. From biological to artificial neural networks

There are literally hundreds of textbooks on neural networks and we have no intention whatsoever of reproducing another such textbook here. What we would like to do is point out those types of neural networks that are essential for modeling intelligent behavior, in particular those which are relevant for autonomous agents, to systems that have to interact with the real world. The goal of this chapter is to provide an intuition rather than a lot of technical detail. The brain consists of roughly 10^{11} neurons. They are highly interconnected, each neuron making up to 10'000 connections, or synapses, with other neurons. This yields roughly 10^{14} synapses. The details do not matter here. We would simply like to communicate a flavor of the awesome complexity of the brain. In fact, it is often claimed that the human brain is the most complex known structure in the universe. More precisely, it's the human organism which contains, as one of its parts, the brain.

Figure 2 (a) shows a model of a biological neuron in the brain. For our purposes we can ignore the physiological processes. The interested reader is referred to the excellent textbooks in the field (e.g. [**Kandel et al., 1995**]).

The main components of a biological neuron are the dendrites which have the task of transmitting activation from other neurons to the cell body of the neuron, which in turn has the task of summing incoming activation, and the axon which will transmit information, depending on the state of the cell body. The information on the cell body's state is transmitted to other neurons via the axons by means of a so-called spike, i.e., an action potential which quickly propagates along an axon. The axon makes connections to other neurons. The dendrites can be excitatory, which means that they influence the activation level of a neuron positively, or they can be inhibitory in which case they potentially decrease the activity of a neuron. The impulses reaching the cell body (soma) from the dendrites arrive asynchronously at any point in time. If enough excitatory impulses arrive within a certain small time interval, the axon will send out signals in the form of spikes. These spikes can have varying frequencies.

This description (Figure 2 (b)) represents a drastic simplification; individual neurons are highly complex in themselves, and almost daily additional properties

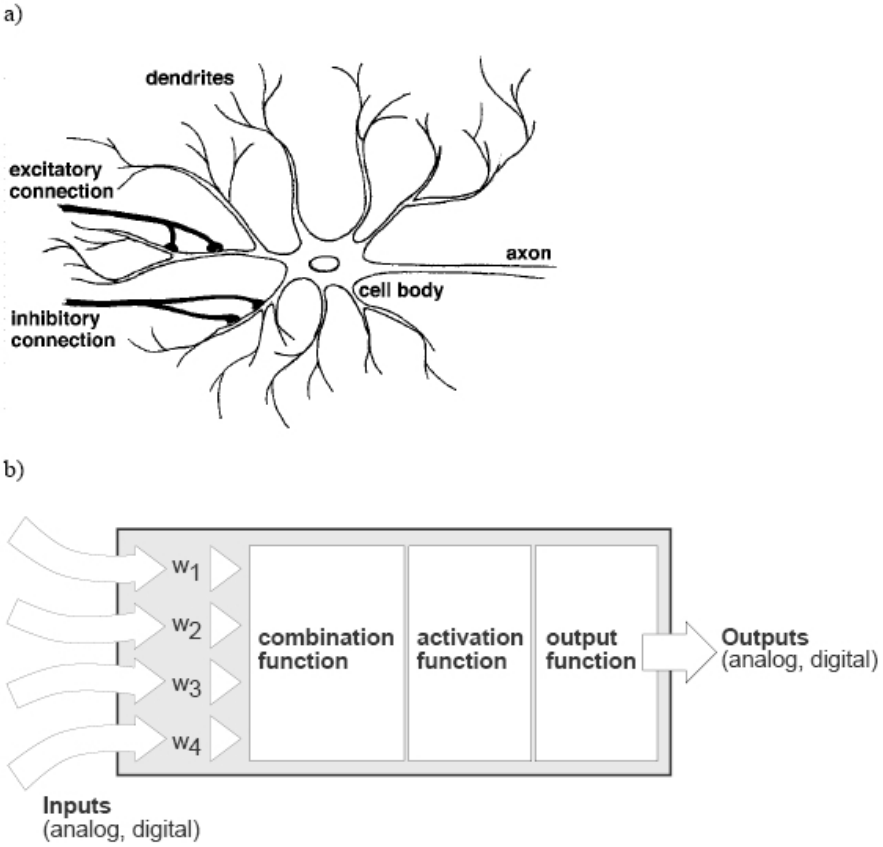


FIGURE 2. Natural and artificial neurons. Model of (a) a biological neuron, (b) artificial neuron. The dendrites correspond to the connections between the cells, the synapses to the weights, the outputs to the axons. The computation is done in the cell body.

are discovered. If we want to develop models even of some small part of the brain we have to make significant abstractions. We now discuss some of them.

One abstraction that is typically made is that there is some kind of a clock which synchronizes all the activity in the network. In this abstraction inputs to an (artificial) neuron can simply be summed to yield a level of activation, whereas to model a real biological system one would have to take the precise arrival times of the incoming signals - the spikes - into account or one would have to assume a statistical distribution of arrival times. Moreover, the spikes are not modeled individually, but only their average firing rate (In chapter 7, we briefly bring up the issue of how one can handle the more detailed properties of spiking neurons). The firing rate is the number of spikes per second produced by the neuron. It is given by one simple output value. An important aspect which is neglected in many ANN models is the amount of time it takes for a signal to travel along the axon. In some architectures such delays are considered explicitly (e.g. [**Ritz and Gerstner, 1994**]). Nevertheless, it is amazing how much can be achieved by

nervous system	*Artificial neural network*
neuron	Processing element, node, model neuron, abstract neuron
dendrites	Connections with propagation rule
cell body	Activation level, activation function, transfer function, output function
spike	Output of a node
axon	Connection to other neurons
synapses	Connection strengths/weights

FIGURE 3. Natural and artificial neural networks
where is the caption??

employing this very abstract model or variations thereof. Table in Figure 3 shows the correspondences between the respective properties of real biological neurons in the nervous system and abstract neural networks.

Before going into the details of neural network models, let us just mention one point concerning the level of abstraction. In natural brains, there are many different types of neurons, depending on the degree of differentiation, several hundred. Moreover, the spike is only one way in which information is transmitted from one neuron to the next, although it is a very important one. (e.g. [**Kandel et al., 1991**], [**Churchland and Sejnowski, 1992**]). Just as natural systems employ many different kinds of neurons and ways of communicating, there is a large variety of abstract neurons in the neural network literature.

Given these properties of real biological neural networks we have to ask ourselves, how the brain achieves its impressive levels of performance on so many different types of tasks. How can we achieve *anything* using such models as a basis for our endeavors? Since we are used to traditional sequential programming this is by no means obvious. In what follows we demonstrate how one might want to proceed. Often, the history of a field helps our understanding. The next section introduces the history of connectionism, a special direction within the field of artificial neural networks, concerned with modeling cognitive processes.

3. The history of connectionism

During the eighties, a new kind of modeling technique or modeling paradigm emerged, connectionism. We already mentioned that the term connectionism is used to designate the field that applies neural networks to modeling phenomena from cognitive science. As we will show in this chapter, by neural networks we mean a particular type of computational model consisting of many, relatively simple, interconnected units working in parallel. Because of the problems of classical approaches to AI and cognitive science, connectionism was warmly welcomed by many researchers. It soon had a profound impact on cognitive psychology and large portions of the AI community. Actually, connectionism was not really new at the time; it would be better to speak of a renaissance. Connectionist models have been around since the 1950s when Rosenblatt published his seminal paper on perceptrons (e.g. [**Rosenblatt, 1958**]). Figure 4 illustrates Rosenblatt's perceptron.

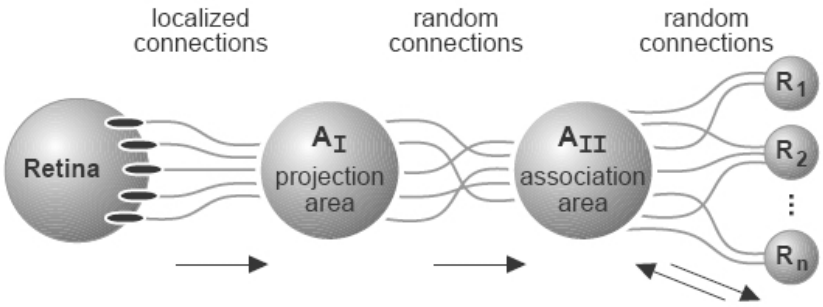


FIGURE 4. Illustration of Rosenblatt's perceptron. Stimuli impinge on a retina of sensory units (left). Impulses are transmitted to a set of association cells, also called the projection area. This projection may be omitted in some models. The cells in the projection area each receive a number of connections from the sensory units (a receptive field, centered around a sensory unit). They are binary threshold units. Between the projection area and the association area, connections are assumed to be random. The responses R_i are cells that receive input typically from a large number of cells in the association area. While the previous connections were feedforward, the ones between the association area and the response cells are both ways. They are either excitatory, feeding back to the cells they originated from, or they are inhibitory to the complementary cells (the ones from which they do not receive signals). Although there are clear similarities to what is called a perceptron in today's neural network literature, the feedback connections between the response cells and the association are normally missing.

Even though all the basic ideas were there, this research did not really take off until the 1980s. One of the reasons was the publication of Minsky and Papert's seminal book "Perceptrons" in 1969. They proved mathematically some intrinsic limitations of certain types of neural networks (e.g. [**Minsky and Papert, 1969**]). The limitations seemed so restrictive that, as a result, the symbolic approach began to look much more attractive and many researchers chose to pursue the symbolic route. The symbolic approach entirely dominated the scene until the early eighties; then problems with the symbolic approach started to come to the fore.

The years between 1985 and 1990 were really the heydays of connectionism. There was an enormous hype and a general belief that we had made enormous progress in our understanding of intelligence. It seems that what the researchers and the public at large were most fascinated with were essentially two properties: First, neural networks are learning systems, and second they have emergent properties. In this context, the notion of emergent properties refers to behaviors a neural network (or any system) exhibits that were not programmed into the system. They result from an interaction of various components among each other (and with the environment, as we will see later). A famous example of an emergent

phenomenon has been found in the NETTalk model, a neural network that learns to pronounce English text (NETTalk will be discussed in chapter 3). After some period of learning, the network starts to behave as if it had learned the rules of English pronunciation, even though there were no rules in the network. So, for the first time, computer models were available that could do things the programmer had not directly programmed into them. The models had acquired their own history! This is why connectionism, i.e. neural network modeling in cognitive science, still has somewhat of a mystical flavor.

Neural networks are now widely used beyond the field of cognitive science (see section 1.4). Applications abound in areas like physics, optimization, control, time series analysis, finance, signal processing, pattern recognition, and of course, neurobiology. Moreover, since the mid-eighties when they started becoming popular, many mathematical results have been proved about them. An important one is their computational universality (see chapter 4). Another significant insight is the close link to statistical models (e.g. [**Poggio and Girosi, 1990**]). These results change the neural networks into something less mystical and less exotic, but no less useful and fascinating.

4. Directions and Applications

Of course, classifications are always arbitrary, but one can identify roughly four basic orientations in the field of neural networks, cognitive science/artificial intelligence, neurobiological modeling, general scientific modeling, and computer science which includes its applications to real-world problems. In cognitive science/artificial intelligence, the interest is in modeling intelligent behavior. This has been the focus in the introduction given above. The interest in neural networks is mostly to overcome the problems and pitfalls of classical - symbolic - methods of modeling intelligence. This is where connectionism is to be located. Special attention has been devoted to phenomena of emergence, i.e. phenomena that are not contained in the individual neurons, but the network exhibits global behavioral patterns. We will see many examples of emergence as we go on. This field is characterized by a particular type of neural network, namely those working with activation levels. It is also the kind mostly used in applications in applied computer science. I is now common practice in the fields of artificial intelligence and robotics to apply insights from neuroscience to the modeling of intelligent behavior.

Neurobiological modeling has the goal to develop models of biological neurons. Here, the exact properties of the neurons play an essential role. In most of these models there is a level of activation of an individual neuron, but the temporal properties of the neural activity (spikes) are explicitly taken into account. Various levels are possible, all the way down to the ion channels to model the membrane potentials. One of the most prominent examples of this type of simulation is Henry Markram's "Blue Brain" project where models at many levels of abstraction are being developed and integrated. The ultimate goal is to simulate a complete brain, the intermediate one to develop a model of an entire cortical column of a rat brain. Needless to say, these goals are extremely ambitious – and we should always keep in mind that behavior is an interaction of a complete organism with the environment; brains are part of embodied systems. By merely studying the brain in isolation, we cannot say much about the role of individual neural circuits for the behavior of the system.

Scientific modeling, of which neurobiological modeling is an instance, uses neural networks as modeling tools. In physics, psychology, and sociology neural networks have been successfully applied. Computer science views neural networks as an interesting class of algorithms that has properties –like noise and fault tolerance, and generalization ability– that make them suited for application to real-world problems. Thus, the gamut is huge.

As pointed out, neural networks are now applied in many areas of science. Here are a few examples:

Optimization: Neural networks have been applied to almost any kind of optimization problem. Conversely, neural network learning can often be conceived as an optimization problem in that it will minimize a kind of error function (see chapter 3).

Control: Many complex control problems have been solved by neural networks. They are especially popular for robot control: Not so much factory robots, but autonomous robots - like humanoids - that have to operate in real world environments that are characterized by higher levels of uncertainty and rapid change. Since biological neural networks have evolved for precisely these kinds of conditions, they are well suited for such types of tasks. Also, because in the real world generalization is crucial, neural networks are often the tool of choice for systems, in particular robots, having interact with physical environments.

Signal processing: Neural networks have been used to distinguish mines from rocks using sonar signals, to detect sun eruptions, and to process speech signals. Speech processing techniques and statistical approaches involving hidden Markov models are sometimes combined.

Pattern recognition: Neural networks have been widely used for pattern recognition purposes, from face recognition, to recognition of tumors in various types of scans, to identification of plastic explosives in luggage of aircraft passengers (which yield a particular gamma radiation patterns when subjected to a stream of thermal neurons), to recognition of hand-written zip-codes.

Stock market prediction: The dream of every mathematician is to develop methods for predicting the development of stock prices. Neural networks, in combination with other methods, are often used in this area. However, at this point in time, it is an open question whether they have been really successful, and if they have, the results wouldn't have been published.

Classification problems: Any problem that can be couched in terms of classification is a potential candidate for a neural network solution. Many have been mentioned already. Examples are: stock market prediction, pattern recognition, recognition of tumors, quality control (is the product good or bad), recognition of explosives in luggage, recognition of hand-written zip codes to automatically sort mail, and so on and so forth. Even automatic driving could be viewed as a kind of classification problem: Given a certain pattern of sensory input (e.g. from a camera, or a distance sensor), which is the best angle for the steering wheel, and the degree of pushing the accelerator or the brakes.

In the subsequent chapters, we systematically introduce the basic concepts and the different types of models and architectures.

CHAPTER 2

Basic concepts

Although there is an enormous literature on neural networks and a very rich variety of networks, learning algorithms, architectures, and philosophies, a few underlying principles can be identified. All the rest consists of variations of these few basic principles. The "four or five basics", discussed here, provide such a simple framework. Once this is understood, it should present no problem to dig into the literature.

1. The four or five basics

For every artificial neural network we have to specify the following four or five basics. There are four basics that concern the network itself. The fifth one – equally important – is about how the neural network is connected to the real world, i.e. how it is embedded in the physical system. Embedded systems are connected to the real world through their own sensory and actuator systems. Because of their properties of robustness, neural networks are well-suited for such types of systems. Initially, we will mostly focus mostly on the computational properties (1) through (4) but later discuss complete embodied systems, in particular robots.

(1) *The characteristics of the node.* We use the terms nodes, units, processing elements, neurons, and artificial neurons synonymously. We have to define the way in which the node sums the inputs, how they are transformed into level of activation, how this level of activation is updated, and how it is transformed into an output which is transmitted along the axon.
(2) *The connectivity.* It must be specified which nodes are connected to which and in what direction.
(3) *The propagation rule.* It must be specified how a given activation that is traveling along an axon, is transmitted to the neurons to which it is connected.
(4) *The learning rule.* It must be specified how the strengths of the connections between the neurons change over time.
(5) *The fifth basic: embedding the network in the physical system*: If we are interested in neural networks for embedded systems we must always specify how the network is embedded, i.e. how it is connected to the sensors and the motor components.

In the neural network literature there are literally thousands of different kinds of network types and algorithms. All of them, in essence, are variations on these basic properties.

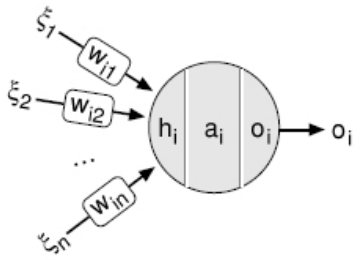


FIGURE 1. Node characteristics. a_i: activation level, h_i: summed weighted input into the node (from other nodes), o_i: output of node (often identical with a_i), w_{ij}: weights connecting nodes j to node i (This is a mathematical convention used in Hertz, Krogh and Palmer, 1991; other textbooks use the reverse notation. Both notations are mathematically equivalent). ξ_i: inputs into the network, or outputs from other nodes. Moreover, with each node the following items are associated: an activation function g, transforming the summed input h_i into the activation level, and a threshold, indicating the level of summed input required for the neuron to become active. The activation function can have various parameters.

2. Node characteristics

We have to specify how the incoming activation is summed, processed to yield level of activation, and how output is generated.

The standard way of calculating the level of activation of a neuron is as follows:

$$a_i = g(\sum_{j=1}^{n} w_{ij} o_j) = g(h_i) \tag{1}$$

where a_i is the level of activation of neuron i, o_j the output of other neurons, g the activation

function, h_i the summed activation, and o_i is the output. Normally we have $o_i = f(a_i) = a_i$, i.e., the output is taken to be the level of activation. In this case, equation (1) can be rewritten as

$$a_i = g(\sum_{j=1}^{n} w_{ij} a_j) = g(h_i) \tag{2}$$

Figure 2 shows the most widely used activation functions. Mathematically speaking the simplest one is the linear function (a). The next one is the step function which is non-linear (b): there is a linear summation of the inputs and nothing happens until the threshold Θ is reached at which point the neuron becomes active (i.e., shows a certain level of activation). Such units are often called *linear threshold* units. The third kind to be discussed here is the *sigmoid* or *logistic* function (c).

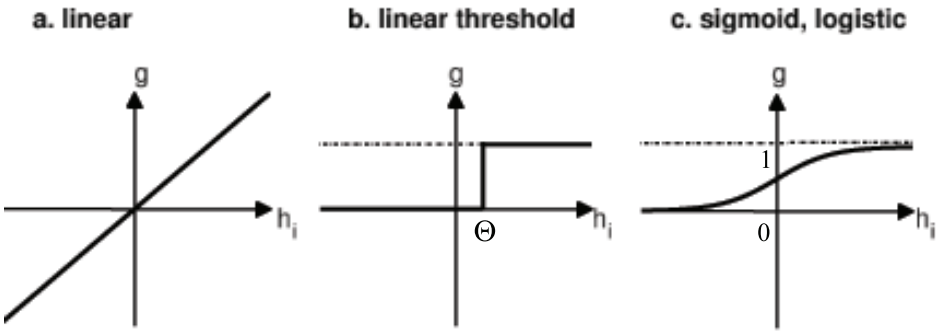


FIGURE 2. Most widely used activation functions. h_i is the summed input, g the activation function. (a) linear function, (b) step function, (c) sigmoid function (also logistic function).

The sigmoid function is, in essence, a smooth version of a step function.

$$g(h_i) = \frac{1}{1 + e^{-2\beta h_i}} \tag{3}$$

with $\beta = 1/k_B T$ (where T can be understood as the absolute temperature). It is zero for low input. At some point it starts rising rapidly and then, at even higher levels of input, it saturates. This saturation property can be observed in nature where the firing rates of neurons are limited by biological factors. The slope, β (also called gain) is an important parameter of the sigmoid function: The larger β, the steeper the slope, the more closely it approximates the threshold function.

The sigmoid function varies between 0 and 1. Sometimes an activation function that varies between -1 and +1 with similar properties is required. This is the hyperbolic tangent:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{4}$$

The relation to the sigmoid function g is given by $\tanh(\beta h) = 2g(h) - 1$. Because in the real world, there are no strict threshold functions, the "rounded" versions – the sigmoid functions – are somewhat more realistic approximations of biological neurons (but they still represent substantial abstractions).

While these are the most frequently used activation functions, others are also used, e.g. in the case of radial basis functions which are discussed later. Radial basis function networks often use Gaussians as activation functions.

3. Connectivity

The second property to be specified for any neural network is the connectivity, i.e., how the individual nodes are connected to one another. This can be done by means of a directed graph with nodes and arcs (arrows). Connections are only in one direction. If they are bi-directional, this must be explicitly indicated by two arrows. Figure 3 shows a simple neural net. Nodes 1 and 2 are input nodes; they receive activation from outside the network. Node 1 is connected to nodes 3, 4, and 5, whereas node 3 is connected to node 1. Nodes 3, 4 and 5 are output nodes. They could be connected to a motor system, where node 3 might stand for "turn left",

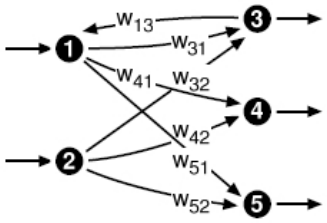


FIGURE 3. Graphical representation of a neural network. The connections are called w_{ij}, meaning that this connection links node j to node i with weight w_{ij} (note that this is intuitively the "wrong" direction, but it is just a notational convention, as pointed out earlier). The matrix representation for this network is shown in Figure 4

	node 1	*node 2*	*node 3*	*node 4*	*node 5*
node 1	0	0	0.8	0	0
node 2	0	0	0	0	0
node 3	0.7	0.4	0	0	0
node 4	1.0	-0.5	0	0	0
node 5	0.6	0.9	0	0	0

FIGURE 4. Matrix representation of a neural network

node 4 for "straight", and node 5 for "turn right". Note that nodes 1 and 3 are connected in both directions, whereas between nodes 1 and 4 the connection is only one-way. Connections in both directions can be used to implement some kind of short-term memory. Networks having connections in both directions are also called recurrent networks (see chapter 5). Nodes that have similar characteristics and are connected to other nodes in similar ways are sometimes called a *layer*. Nodes 1 and 2 receive input from outside the network; they are called the *input layer*, while nodes 3, 4, and 5 form the *output layer*.

For larger networks, the graph notation gets cumbersome and it is better to use matrices. The idea is to list all the nodes horizontally and vertically. The matrix elements are the connection strengths. They are called w_{ij}, meaning that this weight connects node j to node i (note that this is intuitively the "wrong" direction, but it is just a notational convention). This matrix is called the connectivity matrix. It represents, in a sense, the "knowledge" of the network. In virtually all types of neural networks, the learning algorithms work through modification of the weight matrix. However, in some learning algorithms, other parameters of the network are also modified, e.g. the gain of the nodes. Throughout the field of neural networks, matrix notation is used. It is illustrated in Figure 4.

Node 1 is not connected to itself ($w_{11} = 0$), but it is connected to nodes 3, 4, and 5 (with different strengths w_{31}, w_{41}, w_{51}). The connection strength determines how much activation is transferred from one node to the next. Positive connections are excitatory, negative ones inhibitory. Zeroes (0) mean that there is no connection. The numbers in this example are chosen arbitrarily. By analogy to biological neural networks, the connection strengths are sometimes also called *synaptic* strengths. The weights are typically adjusted gradually by means of a learning rule until they are capable of performing a particular task or optimize a particular function (see below). As in linear algebra the term vector is often used in neural network jargon. The values of the input nodes are often called the *input vector*. In the example, the input vector might be (0.6 0.2) (the numbers have again been arbitrarily chosen). Similarly, the list of activation values of the output layer is called the *output vector*. Neural networks are often classified with respect to their connectivity. If the connectivity matrix has all zeroes in the diagonal and above the diagonal, we have *feed-forward* network since in this case there are only forward connections, i.e., connections in one direction (no loops). A network with several layers connected in a forward way is called a *multi-layer feed-forward network* or *multi-layer perceptron*. The network in figure 3 is mostly *feed-forward* (connections only in one direction) but there is one loop in it (between nodes 1 and 3). Loops are important for the dynamical properties of the network. If all the nodes from one layer are connected to all the nodes of another layer we say that they are *fully connected*. Networks in which all nodes are connected to each other in both directions but not to themselves are called *Hopfield* nets (Standard Hopfield nets also have symmetric weights, see later).

4. Propagation rule

We already mentioned that the weight determines how much activation is transmitted from one node to the next. The propagation rule determines how activation is propagated through the network. Normally, a weighted sum is assumed. For example, if we call the activation of node 4 a_4 , we have $a_4 = a_1 w_{41} + a_2 w_{42}$ or generally

$$h_i = \sum_{j=1}^{n} w_{ij} a_j \tag{5}$$

where n is the number of nodes in the network, h_i the summed input to node i. h_i is sometimes also called the *local field* of node i. To be precise, we would have to use o_j instead of a_j., but because the output of the node is nearly always taken to be its level of activation, this amounts to the same thing. This propagation rule is in fact so common that it is often not even mentioned. Note that there is an underlying assumption here, that activation transfer across the links takes exactly one unit of time. We want to make the propagation rule explicit because if - at some point - we intend to model neurons more realistically, we have to take the temporal properties of the propagation process such as delays into account.

In more biologically plausible models, temporal properties of the individual spikes are sometimes taken into account. In these models, it is argued that the information in neural processing lies not only in the activation level (roughly corresponding to the firing rate), but also in the temporal sequences of the spikes, i.e. in

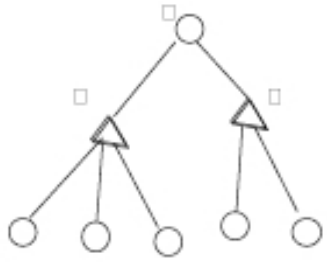


FIGURE 5. Sigma-pi units

the intervals between the spikes. Moreover, traversing a link takes a certain amount of time and typically longer connections require more time to traverse. Normally, unless we are biologically interested, we assume synchronized networks where the time to traverse one link is one time step and at each time step the activation is propagated through the net according to formula 5.

Another kind of propagation rule uses multiplicative connections instead of summation only, as shown in formula 6 (see figure 5). Such units are also called sigma-pi units because they perform a kind of and/or computation. Units with summation only are called sigma units.

$$h_i = \sum w_{ij} \prod w_{ik} a_j \tag{6}$$

There is a lot of work on dendritic trees demonstrating that complicated kinds of computation can already be performed at this level. Sigma-pi units are only a simple instance of them.

While conceptually and from a biological perspective it is obvious tha we have to specify the propagation rule, man textbooks do not deal with this issue explicitly – as mentioned, the one-step assumption is often implicitly adopted.

5. Learning rules

As already pointed out, weights are modified by learning rules. The learning rules determine how "experiences" of a network exert their influence on its future behavior. There are, in essence, three types of learning rules: supervised, reinforcement, and non-supervised or unsupervised.

5.1. Supervised learning. The term supervised is used both in a very general and narrow technical sense. In the narrow technical sense supervised means the following. If for a certain input the corresponding output is known, the network is to learn the mapping from inputs to outputs. In supervised learning applications, the correct output must be known and provided to the learning algorithm. The task of the network is to find the mapping. The weights are changed depending on the magnitude of the error that the network produces at the output layer: the larger the error, i.e. the discrepancy between the output that the network produces – the actual output – and the correct output value – the desired output –, the more the weights change. This is why the term *error-correction learning* is also used.

Examples are the *perceptron learning rule*, the *delta rule*, and - most famous of all - *backpropagation*. Back-propagation is very powerful and there are many variations of it. The potential for applications is enormous, especially because such networks have been proved to be universal function approximators. Such learning algorithms are used in the context of feedforward networks. Back-propagation requires a multi-layer network. Such networks have been used in many different areas, whenever a problem can be transformed into one of classification. A prominent example is the recognition of handwritten zip codes which can be applied to automatically sorting mail in a post office. Supervised networks will be discussed in great detail later on.

There is also a non-technical use of the word supervised. In a non-technical sense it means that the learning, say of children, is done under the supervision of a teacher who provides them with some guidance. This use of the term is very vague and hard to translate into concrete neural network algorithms.

5.2. Reinforcement learning. If the teacher only tells a student whether her answer is correct or not, but leaves the task of determining why the answer is correct or false to the student, we have an instance of *reinforcement learning*. The problem of attributing the error (or the success) to the right cause is called the credit assignment or blame assignment problem. It is fundamental to many learning theories. There is also a more technical meaning of the term of *reinforcement learning* as it is used in the neural network literature. It is used to designate learning where a particular behavior is to be reinforced. Typically, the robot receives a positive reinforcement signal if the result was good, no reinforcement or a negative reinforcement signal if it was bad. If the robot has managed to pick up an object, has found its way through a maze, or if it has managed to shoot the ball into the goal, it will get a positive reinforcement. Reinforcement learning is not tied to neural networks: there are many reinforcement learning algorithms in the field of machine learning in general. To use Andy Barto's words, one of the champions of reinforcement learning "Reinforcement learning [...] is a computational approach to learning whereby an agent tries to maximize the total amount of reward it receives when interacting with a complex, uncertain environment."([**Sutton and Barto, 1998**])

5.3. Unsupervised learning. Mainly two categories of learning rules fall under this heading: Hebbian learning and competitive learning. Hebbian learning which we will consider in a later chapter establishes correlations: if two nodes are active simultaneously (or within some time window) the connection between them is strengthened. Hebbian learning has become popular because - though it is not very powerful as a learning mechanism - it requires only local information and there is a certain biological plausibility to it. Hebbian learning is closely related to spike-time-dependent plasticity, where the change of the synaptic strenght depends on the precise timing of the pre-synaptic and post-synaptic activity of the neuron. In industrial applications, Hebbian learning is not used. Competitive learning, in particular Kohonen networks are used to find clusters in data sets. Kohonen networks also have a certain biological plausibility. In addition, they have been put to many industrial usages. We will discuss Kohonen networks in detail later.

6. The fifth basic: embedding the network

If we want to use a neural network for controlling a physical device, it is quite obvious that we have to connect it to the device. We have to specify how the sensory signals are going to influence the network and how the computations of the network are going to influence the device's behavior. In other words we must know about the physics of the sensors and the motor system. This is particularly important if we have the goal of understanding biological brains or if we want to use neural networks to control robots. In this course we will use many examples of robots and so we must always keep the fifth basic in mind.

CHAPTER 3

Simple perceptrons and adalines

1. Historical comments and introduction

This section introduces two basic kinds of learning machines from the class of supervised models: perceptrons and adalines. They are similar but differ in their activation function and as a consequence, in their learning capacity. We start with a historical comment, introduce classification, perceptron learning rules, adalines and delta rules.

2. The perceptron

The perceptron goes back to Rosenblatt (1958), as described in chapter 1. He formulated a learning algorithm, a way to systematically change the weights and proved its convergence, i.e. that after a finite number of steps, the perceptron would perform the desired computation. This generated a lot of enthusiasm in the community and there was the hope that machines would soon be capable of learning virtually anything.

There was, however, a limitation in Rosenblatt's learning theorem, as pointed out by Marvin Minsky (one of the founders of the field of artificial intelligence) and Seymour Papert (the inventor of the "Logo Turtles"), both mathematicians at MIT. They demonstrated in their book "Perceptrons" that the theorem – obviously – only applies to those problems whose solutions can actually been computed. They showed that perceptrons could not perform some seemingly simple computations, a prominent example being the famous XOR problem: Given a network with two input nodes and one output node, if the input is zero and one, the output should be one, if both inputs are one or both are zero, the output should be zero.

In order to overcome the limitations, Rosenblatt also studied networks with several layers – which was the right intuition! However, at the time, there was no learning mechanism available to determine the weights to perform the desired computation. Because Minsky and Papert were skeptical about whether such learning schemes could be found, they and many others turned to symbol computation and the field of artificial intelligence turned into a discipline largely based on symbol processing. So, perceptrons, for a period of almost 20 years, largely disappeared from the scene. Still, a number of people continued working on perceptron-like or more generally network-like ideas.

But it was not until the 1980s when learning mechanisms were discovered that work in networks with multiple layers, the most famous of them being error back-propagation, e.g. by Rumelhart, Hinton and Williams in 1986 ([**Rumelhart et al., 1988**]). Some people say that the learning algorithms had been re-discovered because Werbos had published similar results already in 1974 ([**Werbos, 1974a**]). However that may be, the availability of learning algorithms for multilayer networks

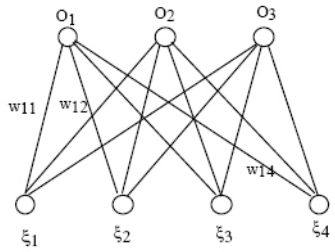


FIGURE 1. A perceptron with 4 input units and 3 output units. Often the inputs and outputs are labeled with a pattern index μ, and patterns typically range from 1 to p. O_i^μ is used to designate the actual output of the network. ζ_i^μ designates the desired output.

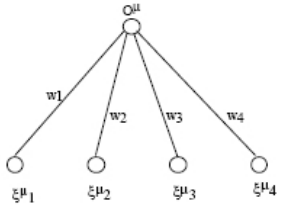


FIGURE 2. A simplified perceptron with only one output node.

led to a real explosion in the field of neural networks. Since then, neural networks have pervaded artificial intelligence, psychology, and the cognitive sciences.

2.1. The classification problem. Before we define the classification problem, we need to introduce a bit of linear algebra (see also linear algebra tutorial).

Let $\underline{\xi}$ be the input vector and g the activation function (binary threshold)

Figure 2 shows a simplified network with only one output node.

$$\underline{\xi} = (\xi_1, \xi_2, \ldots, \xi_n)^T = \begin{pmatrix} \xi_1 \\ \xi_2 \\ \cdots \\ \cdots \\ \xi_n \end{pmatrix}$$

$$g : O = 1, \text{if } \underline{w}^T \underline{\xi} = \sum_{j=1}^{n} w_j \xi_j \geq \Theta$$

$$g : O = 0, \text{if } \underline{w}^T \underline{\xi} = \sum_{j=1}^{n} w_j \xi_j < \Theta$$

(7) $\underline{w}^T \underline{\xi}$: scalar product, inner product

Let us now define the classification problem for perceptrons. Let $\Omega = \{\xi\}$ the set of all possible patterns that the perceptron network should learn to classify.

$\Omega = \Omega_1 \cup \Omega_2$
ξ in Ω_1 : O should be 1 (true)
ξ in Ω_2 : O should be 0 (false) (alternatively it could be -1)

Alternatively – and also often used in the literature – the desired output should be -1. We will be using both notations as we go along.

Learning goal: Learn separation such that for each ξ in $\Omega_1 \sum_{j=1}^{n} w_j \xi_j \geq \Theta$ and for each ξ in $\Omega_2 \sum_{j=1}^{n} w_j \xi_j < \Theta$

Example: AND problem

Let us look at a very simple example, the AND problem.

$\Omega_1 = \{(1,1)\} \rightarrow 1$

$\Omega_2 = \{(0,0),(0,1),(1,0)\} \rightarrow 0$ Learning goal: The network should learn to produce a 1 at its output if the input vector is $(1,1)$ and 0 if it is $(0,0)$,$(0,1)$, or $(1,0)$. If the output is correct, we don't change anything. If it is incorrect we have to distinguish two cases, i.e. $O = 1$, but should be 0, and $O = 0$ but should be 1. Let's try to get an intuition of the perceptron learning rule by looking at how thresholds and weights have to be changed:

1. Thresholds:
O=1, should be 0: $\Theta \rightarrow$ increase
O=0, should be 1: $\Theta \rightarrow$ decrease

2. Weights:
O=1, should be 0:
if $\xi_i = 0 \rightarrow$ no change
if $\xi_i = 1 \rightarrow$ decrease w_i
O=0, should be 1:
if $\xi_i = 0 \rightarrow$ no change
if $\xi_i = 1 \rightarrow$ increase w_i

Trick:
$\xi \rightarrow (-1, \xi_1, \xi_2, \ldots, \xi_n)$
$w \rightarrow (\Theta, w_1, w_2, \ldots, w_n)$
Notation: $(\xi_0, \xi_1, \xi_2, \ldots, \xi_n); (w_0, w_1, w_2, \ldots, w_n)$

2.2. Perceptron learning rule. Since most of the knowledge in a neural network is contained in the weights, learning means systematically changing the weights. There are also learning schemes where other parameters (e.g. the gain of the activation function) are changed – we will look at a few examples later in the course, but for now, we focus on weight change.

$$\begin{gathered} \xi \text{ in } \Omega_1 \\ \text{if } w^T \xi \geq 0 \rightarrow \text{OK} \\ \text{if } w^T \xi < 0 \\ \underline{w}(t) = \underline{w}(t-1) + \eta \underline{\xi} \\ w_i(t) = w_i(t-1) + \eta \xi_i \end{gathered} \tag{8}$$

ξ_1	ξ_2	ζ /AND
0	0	0
0	1	0
1	0	0
1	1	1

FIGURE 3. Truth table for the AND function

$$\begin{gathered} \xi \text{ in } \Omega_2 \\ \text{if } w^T\xi \geq 0 \\ \underline{w}(t) = \underline{w}(t-1) - \eta\underline{\xi} \\ w_i(t) = w_i(t-1) - \eta\xi_i \\ \text{if } w^T\xi < 0 \rightarrow \text{OK} \end{gathered} \tag{9}$$

Formulas (8 and 9) are a compact way of writing the perceptron learning rule. It includes the thresholds as well as the weights.

Example: $i = 0 \rightarrow w_0$ is threshold

$w_0(t) = w_0(t-1) + \eta\xi_0 \rightarrow -\Theta(t) = -\Theta(t-1) + \eta 1 \rightarrow \Theta(t) = \Theta(t-1) - \eta 1$ (reduction).

The question we then immediately have to ask is under what conditions this learning rule converges. The answer is provided by the famous perceptron convergence theorem:

2.3. Perceptron convergence theorem. The algorithm with the perceptron learning rule terminates after a finite number of iterations with constant increment (e.g. $\eta = 1$), if there is a weight vector $\underline{w}^*$ which separates both classes (i.e. if there is a configuration of weights for which the classification is correct). The next question then is when such a weight vector $\underline{w}^*$ exists. The answer is that the classes have to be *linearly separable*. So, we always have to distinguish clearly between what can be represented and what can be learned (and under what conditions). The fact that something can be represented, e.g. a weight vector that separates the two categories exists, does not necessarily imply that it can also be found by a learning procedure.

Linear separability means that a plane (or if we are talking about high-dimensional spaces, a hyperplane) can be found in the ξ-space separating the patterns in Ω_1 for which the desired value is $+1$, and the patterns in Ω_2 for which the desired value is 0. If there are several output units, such a plane must be found for each output unit. The truth table for the AND function is shown in figure 3.

It is straightforward to formulate the inequalities for the AND problem. Fig. 4 depicts a simple perceptron representing the AND function together with a representation of the input space. The line separating Ω_1 and Ω_2 in 2D input space is given by the equation

$$w_1\xi_1 + w_2\xi_2 = \theta \tag{10}$$

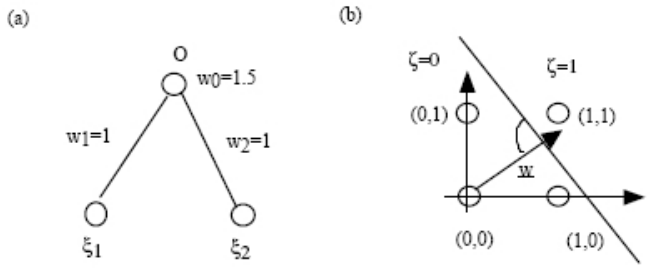


FIGURE 4. A simple perceptron. A possible solution to the AND problem is shown. There is an infinite number of solutions.

ξ_1	ξ_2	ζ /XOR
0	0	0
0	1	1
1	0	1
1	1	0

FIGURE 5. Truth table for the XOR function

which implies

$$\xi_2 = \frac{\theta}{w_2} - \frac{w_1}{w_2}\xi_1. \tag{11}$$

This is the usual form of the equation for a line $y = b + ax$ with slope a and offset b.

And then do the same for the XOR problem (figure 5). As you will see, this latter problem is not linearly separable, in other words, there is no way to satisfy all the inequalities by proper choice of θ, w_1 and w_2.

Pseudocode for the perceptron algorithm:

Select random weights $\underline{w}$ at time t=0.
`.REPEAT`
.Select a random pattern ξ from $\Omega_1 \cup \Omega_2$,t=t+1;
. `IF` (ξ from Ω_1)
. `THEN IF` $\underline{w}^T\underline{\xi} < 0$
. `THEN` $\underline{w}(t) = \underline{w}(t-1) + \eta\underline{\xi}$
. `ELSE` $\underline{w}(t) = \underline{w}(t-1) \rightarrow$ OK
. `ELSE IF` $\underline{w}^T\underline{\xi} \geq 0$
. `THEN` $\underline{w}(t) = \underline{w}(t-1) - \eta\underline{\xi}$
. `ELSE` $\underline{w}(t) = \underline{w}(t-1)$
`.UNTIL` (all $\underline{\xi}$ have been classified correctly)

There are a number of proofs of the perceptron convergence theorem in the literature, for example
Hertz, Krogh, and Palmer ([**Hertz et al., 1991**]), p. 100-101; Minsky and Papert ([**Minsky and Papert, 1969**]); Arbib, 1987, and others. The interested reader

is referred to these publications.

3. Adalines

The Adaline, the Adaptive Linear Element, is very similar to the perceptron: it's also a one-layer feedforward network, the only difference is the activation function at the output, which is linear instead of a step function.

Linear units:
$O_i^\mu = \sum_j w_{ij}\xi_j^\mu$
which implies that O_i^μ is continuous. As usual, the desired output is $O_i^\mu = \zeta_i^\mu$

One advantage of continuous units is that a cost function can be defined. Cost is defined in term of the error, $E(\underline{w})$. This implies that optimization techniques (like gradient methods) can be applied.

3.1. Delta learning rule. Remember that in the perceptron learning rule the factor $\eta\underline{\xi}$ only depends on the input vector and not on the size of the error (because the output is either correct or not). The delta rule takes the size of the error into account.

$$\Delta w_{ij} = -\eta(O_i^\mu - \zeta_i^\mu)\xi_j^\mu = -\eta\delta_i^\mu\xi_j^\mu, \text{ with } \delta_i^\mu = (O_i^\mu - \zeta_i^\mu) \tag{12}$$

This formula is also called the Adaline rule (Adaline=Adaptive linear element) or the Widrow-Hoff rule. If the input is large, the corresponding weights contribute more to the error than if the input is small. In other words, this rule solves the blame assignment problem, i.e. which weights contribute most to the error (and thus need to be changed most by the learning rule). The delta rule implements an LMS procedure (LMS=least mean square), as will be shown, below. Let us define a cost function or error function:

$$\begin{aligned} E(\underline{w}) &= \frac{1}{2}\sum_\mu\sum_i(O_i^\mu - \zeta_i^\mu)^2 = \frac{1}{2}\sum_\mu\sum_i(\sum_j w_{ij}\xi_j^\mu - \zeta_i^\mu)^2 \\ &= \frac{1}{2}\sum_{\mu,i}(\sum_j w_{ij}\xi_j^\mu - \zeta_i^\mu)^2 \end{aligned} \tag{13}$$

where i is the index of the output units and μ runs over all patterns. The better our choice of w's for a given set of input patterns, the smaller E will be. E depends on the weights and on the inputs.

Consider now the weight space (in contrast to the state space which is concerned with the activation levels).

Gradient descent algorithms work as follows: Change each weight w_{ij} by an amount proportional to the negative gradient (the gradient always points up, we want to go down).

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} \tag{14}$$

We can also write the vector notation

$$\Delta \underline{w} = -\eta \underline{\nabla} E(\underline{w}) \tag{15}$$

where the Nabla operator $\underline{\nabla}$ indicates the gradient:

$$\underline{\nabla} E(\underline{w}) = \left(\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_n} \right) \tag{16}$$

The intuition is that we should change the weights in the direction where the error gets smaller the fastest - this is precisely the opposite of the gradient ($-\underline{\nabla}$) in this error function. Using the chain rule and considering that w_{ij} is the only weight which is not "constant" for this operation (which implies that many terms drop out), we get

$$-\eta \frac{\partial E}{\partial w_{ij}} = -\eta \sum_{\mu} (O_i^\mu - \zeta_i^\mu) \xi_j^\mu$$

If we consider one single pattern μ:

$$\Delta w_{ij}^\mu = -\eta (O_i^\mu - \zeta_i^\mu) \xi_j^\mu = -\eta \delta_i^\mu \xi_j^\mu$$

which corresponds to the delta rule. In other words the delta rule realizes a gradient descent procedure in the error function.

3.2. Explicit Solution: (The mathematics of this and the next section is not subject to the final examination. What is important here are the conditions under which solutions exist or can be explicitly calculated in the general case, as well as the generalized delta rule for non-linear units.) In some cases, we can calculate the weights explicitly, i.e. without the need of an iterative learning procedure:

$$w_{ij} = \frac{1}{N} \sum_{\mu,\nu} \zeta_i^\mu (Q^{-1})_{\mu\nu} \xi_j^\nu, \text{where}$$

$$Q_{\mu\nu} = \frac{1}{N} \sum_j \xi_j^\mu \xi_j^\nu \tag{17}$$

$Q_{\mu\nu}$ only depends on the input patterns. Note that we can only calculate the weights in this manner if Q^{-1} exists. This condition requires that the input patterns be *linearly independent*. Linear independence means that there is no set of patterns a_i where not all $a_i = 0$,such that

$$a_1 \xi_j^1 + a_2 \xi_j^2 + \ldots + a_p \xi_j^p = 0, \ \forall j \tag{18}$$

Stated differently, no linear combination of the input vectors can be found such that they add up to 0. The point is that if the input vectors are linearly dependent,

the outputs cannot be chosen independently and then the problem can normally not be solved.
Note that *linear independence* for linear (and non-linear) units is distinct from *linear separability* defined for threshold units (in the case of the classical perceptron). Linear independence implies linear separability, but the reverse is not true. In fact, most of the problems of interest in neural networks do not satisfy the linear independence condition, because the number of patterns is typically larger than the number of dimensions of the input space (i.e. the number of input nodes), i.e. $p > N$. If the number of vectors is larger than the dimension, they are always linearly dependent.

3.3. Existence of solution. Again, we have to ask ourselves when a solution exists. The question "Does a solution exist" means: Is there a set of weights w_{ij} such that all the $\underline{\xi}^\mu$ can be learned such that the actual output is equal to the desired output? "Can be learned" means:

$$O_i^\mu = \zeta_i^\mu, \forall \xi^\mu$$

In other words, the actual output of the network is equal to the desired output for all the patterns $\underline{\xi}^\mu$.

Linear units:
For linear units this is the same as saying:

$$\zeta_i^\mu = \sum_j w_{ij} \xi_j^\mu, \forall \xi^\mu$$

Since Adalines use linear units, the O_i^μ are continuous-valued, which is an advantage over the Perceptron which has only binary output. Still Adalines can only find a solution for classification tasks if the input classes are *linearly separable*.

Non-linear units:
For non-linear units we have to generalize the delta rule, because the latter has been defined for linear units only. This is straightforward. We explain it here because we will need it in the next chapter when introducing the backpropagation learning algorithm.
Assume that g is the standard sigmoid activation function:
$g(h) = [1 + \exp(-2\beta h)]^{-1} = \frac{1}{1+e^{-2\beta h}}$
Note that g is continuously differentiable, which is a necessary property for the generalized delta rule to be applicable.
The error function is:

$$E(\underline{w}) = \frac{1}{2} \sum_{i,\mu} (O_i^\mu - \zeta_i^\mu)^2 = \frac{1}{2} \sum_{i,\mu} [g(h_i^\mu) - \zeta_i^\mu]^2$$

$$h_i^\mu = \sum_j w_{ij} \xi_j^\mu \tag{19}$$

In order to calculate the weight change we form the gradient, just as in the linear case, using the chain rule:

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}}$$

$$\frac{\partial E(\underline{w})}{\partial w_{ij}} = \sum_{\mu} [g(h_i^\mu) - \zeta_i^\mu] g'(h_i^\mu) \xi_j^\mu = \sum_{\mu} \delta_i^\mu \xi_j^\mu$$

$$\delta = [g(h_i^\mu) - \zeta_i^\mu] g'(h_i^\mu)$$

$$\Delta w_{ij} = -\eta \sum_{\mu} \delta_i^\mu \xi_j^\mu \text{ or for a single pattern: } \Delta w_{ij}^\mu = -\eta \delta_i^\mu \xi_j^\mu \tag{20}$$

This *generalized delta rule* now simply contains the derivative of the activation function g'. Because of the specific mathematical form, these derivatives are particularly simple:

$$g(h) = [1 + exp(-2\beta h)]^{-1}$$

$$g'(h) = 2\beta g(1-g) \tag{21}$$

$$\text{for } \beta = \frac{1}{2} \rightarrow g'(h) = g(1-g)$$

We will make use of these relationships in the actual algorithms for back-propagation.
With non-linear activation functions we can finally solve nonlinear problems, so they do not have to be linearly separable anymore. The existence of a solution however, is always different from the question whether a solution *can be found*. We will not go into the details here, but simply mention that in the non-linear case there may be local minima in the error function, whereas in the linear case the global minimum can always be found. The capacity of one-layer perceptrons to represent functions is limited. As long as we have linear units, adding additional layers does not extend the capacity of the network. However if we have non-linear units, the networks become in fact *universal function approximators* (see next chapter).

3.4. Terminology. Before going on to discuss multi-layer perecpetrons, we need to introduce a bit of terminology.
Cycle: 1 pattern presentation, propagate activation through network, change weights
Epoch: 1 "round" of cycles through all the patterns
Error surface: The surface spanned by the error function, plotted in weight space. Given a particular set of patterns $\underline{\xi}^\mu$ to be learned, we have to choose the weights such that the overall error becomes minimal. The error surface visualizes this idea. The learning process can then be viewed as a trajectory on the error surface (see figure 5 in chapter 4).
If the weights are updated only at the end of an epoch, we have the following learning rule which sums over all patterns μ:

$$\Delta w_{ij} = -\eta \sum_{\mu} \delta_i^\mu \xi_j^\mu \tag{22}$$

This is called the *off-line* version. In the off-line version the order in which the patterns appear is irrelevant.

$$\Delta w_{ij}^{\mu} = -\eta \delta_i^{\mu} \xi_j^{\mu} \tag{23}$$

(23) is called the *on-line* version. Here, the weights are updated after each pattern presentation. In this case the order in which the patterns are presented to the network matters.
We will now turn to multi-layer feedforward networks.

CHAPTER 4

Multilayer perceptrons and backpropagation

Multilayer feed-forward networks, or multilayer perceptrons (MLPs) have one or several "hidden" layers of nodes. This implies that they have two or more layers of weights. The limitations of simple perceptrons do not apply to MLPs. In fact, as we will see later, a network with just one hidden layer can represent any Boolean function (including the XOR which is, as we saw, not linearly separable). Although the power of MLPs to represent functions has been recognized a long time ago, only since a learning algorithm for MLPs, backpropagation, has become available, have these kinds of networks attracted a lot of attention. Also, on the theoretical side, the fact that it has been proved in 1989 that, loosely speaking, MLPs are universal function approximators [**Hornik et al., 1989**], has added to their visibility (but see also section 4.6).

1. The back-propagation algorithm

The back-propagation algorithm is central to much current work on learning in neural networks. It was independently invented several times (e.g. [**Bryson and HO, 1969**, **Werbos, 1974b**, **Rumelhart et al., 1986b**, **Rumelhart et al., 1986a**])

As usual, the patterns are labeled by μ, so input k is set to ξ_k^μ when pattern μ is presented. The ξ_k^μ can be binary (0,1) or continuous-valued. As always, N

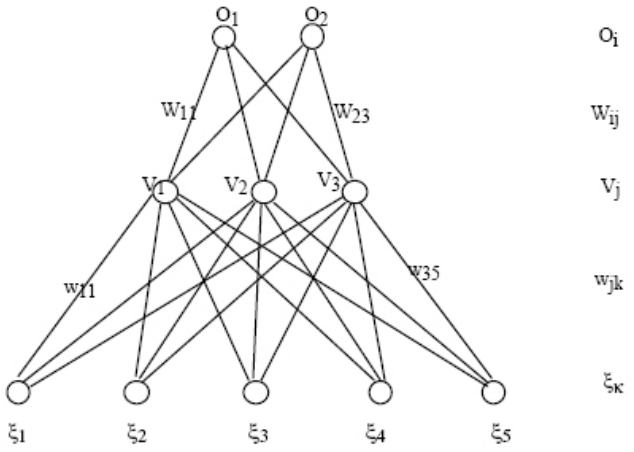


FIGURE 1. A two-layer perceptron showing the notation for units and weights.

designates the number of input units, p the number of input patterns (μ =1, 2, ..., p).
For an input pattern μ, the input to node j in the hidden layer (the V-layer) is

$$h_j^\mu = \sum_k w_{jk}\xi_k^\mu \tag{24}$$

and the activation of the hidden node V_j^μ becomes

$$V_j^\mu = g(h_j^\mu) = g(\sum_k w_{jk}\xi_k^\mu) \tag{25}$$

where g is the sigmoid activation function. Output unit i (O-layer) gets

$$h_i^\mu = \sum_j W_{ij}V_j^\mu = \sum_j W_{ij}g(\sum_k w_{jk}\xi_k^\mu) \tag{26}$$

and passing it through the activation function g we get:

$$O_i^\mu = g(h_i^\mu) = g(\sum_j W_{ij}V_j^\mu) = g(\sum_j W_{ij}g(\sum_k w_{jk}\xi_k^\mu)) \tag{27}$$

Thresholds have been omitted. They can be taken care of by adding an extra input unit, the bias node, connecting it to all the nodes in the network and clamping it's value to (-1); the weights from this unit represent the thresholds of each unit.
The error function is again defined for all the output units and all the patterns:

$$\begin{aligned} E(\underline{w}) &= \frac{1}{2}\sum_{\mu,i}[O_i^\mu - \zeta_i^\mu]^2 = \frac{1}{2}\sum_{\mu,i}[g(\sum_j W_{ij}V_j^\mu) - \zeta_i^\mu]^2 \\ &= \frac{1}{2}\sum_{\mu,i}[g(\sum_j W_{ij}g(\sum_k w_{jk}\xi_k^\mu)) - \zeta_i^\mu]^2 \end{aligned} \tag{28}$$

Because this is a continuous and differentiable function of every weight we can use a gradient descent algorithm to learn appropriate weights:

$$\Delta W_{ij} = -\eta\frac{\partial E}{\partial W_{ij}} = -\eta\sum_\mu[O_i^\mu - \zeta_i^\mu]g'(h_i^\mu)V_j^\mu = -\eta\sum_\mu \delta_i^\mu V_j^\mu \tag{29}$$
$$\text{with } \delta_i^\mu = [O_i^\mu - \zeta_i^\mu]g'(h_i^\mu)$$

To derive (29) we have used the following relation:

$$\begin{aligned} &\frac{\partial}{\partial W_{ij}}g(\sum_j W_{ij}g(\sum_k w_{jk}\xi_k^\mu)) = g'(h_i^\mu)V_j^\mu \\ &W_{11}V_1^\mu + W_{12}2_1^\mu + \ldots + W_{ij}V_j^\mu + \ldots \end{aligned} \tag{30}$$

Because the W_{11} etc. are all constant for the purpose of this differentiation, the respective derivatives are all 0, except for W_{ij}.
And

$$\frac{\partial W_{ij} V_j^\mu}{\partial W_{ij}} = V_j^\mu$$

As noted in the last chapter, for sigmoid functions the derivatives are particularly simple:

$$g'(h_i^\mu) = O_i^\mu (1 - O_i^\mu)$$
$$\delta_i^\eta = (O_i^\mu - \zeta_i^\mu) O_i^\mu (1 - O_i^\mu) \tag{31}$$

In other words, the derivative can be calculated from the function values only (no derivatives in the formula any longer)!
Thus for the weight changes from the hidden layer to the output layer we have:

$$\Delta W_{ij} = -\eta \frac{\partial E}{\partial W_{ij}} = -\eta \sum_\mu \delta_i^\mu V_j^\mu = -\eta \sum_\mu (O_i^\mu - \zeta_i^\mu) O_i^\mu (1 - O_i^\mu) V_j^\mu \tag{32}$$

Note that g no longer appears in this formula.
In order to get the derivatives of the weights from input to hidden layer we have to apply the chain rule:

$$\Delta w_{jk} = -\eta \frac{\partial E}{\partial w_{jk}} = -\eta \frac{\partial E}{\partial V_j^\mu} \frac{\partial V_j^\mu}{\partial w_{jk}}$$

after a number of steps we get:

$$\Delta w_{jk} = -\eta \sum_\mu \delta_j^\mu \xi_k^\mu$$
$$\text{where } \delta_j^\mu = (\sum_i W_{ij} \delta_i^\mu) g'(h_j^\mu) = (\sum_i W_{ij} \delta_i^\mu) V_j^\mu (1 - V_j^\mu) \tag{33}$$

And here is the complete algorithm for backpropagation:

Naming conventions:

m: index for layer
M: number of layers
$m = 0$: input layer
$V_i^0 = \xi_i$; *weight* w_{ij}^m: V_j^{m-1} *to* V_i^m; ζ_i^μ: desired output

(1) Initialize weights to small random numbers.
(2) Choose a pattern ξ_k^μ from the training set; apply to input layer (m=0) $V_k^0 = \xi_k^\mu$ for all k
(3) Propagate the activation through the network:
$V_i^m = g(h_i^m) = g(\sum_j w_{ij}^m V_j^{m-1})$
for all i and m until all V_i^M have been calculated (V_i^M = activations of the units of the output layer).

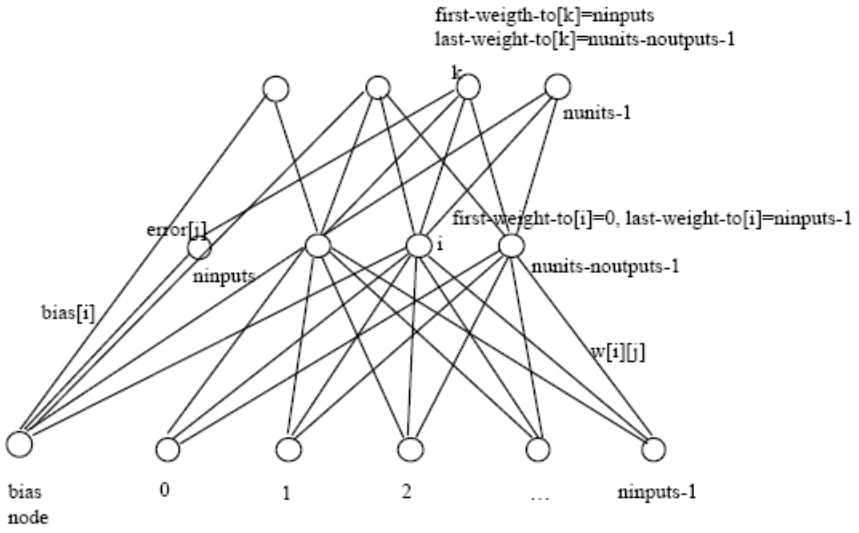


FIGURE 2. Illustration of the notation for the backpropagation algorithm.

(4) Compute the deltas for the output layer M:
$\delta_i^M = g'(h_i^M)[\zeta_i^\mu - V_i^M]$,
for sogmoid: $\delta_i^M = V_i^M(1 - Vi^M)[\zeta_i^\mu - V_i^M]$
(5) Compute the deltas for the preceding layers by successively propagating the errors backwards
$\delta_i^{m-1} = g'(h_i^{m-1}) \sum_j w_{ji}^m \delta_j^m$
for m=M, M-1, M-2, ..., 2 until a delta has been calculated for every unit.
(6) Use
$\Delta w_{ij}^m = \eta \delta_i^m V_j^{m-1}$
to update all connections according to
(*) $w_{ij}^{new} = w_{ij}^{old} + \Delta w_{ij}$
(7) Go back to step 2 and repeat for the next pattern.

Remember the distinction between the "on-line" and the "off-line" version. This is the on-line version because in step 6 the weights are changed after each individual pattern has been processed (*). In the off-line version, the weights are changed only once all the patterns have been processed, i.e. (*) is executed only at the end of an epoch. As before, only in the online version the order of the patterns matters.

2. Java-code for back-propagation

Figure 2 demonstrates the naming conventions in the actual Java-code. This is only one possibility - as always there are many ways in which this can be done.

Shortcuts:

i++	Increase index i by 1
i–	Decrease index i by 1
+=	Increase value of the variable

Variables

first_weight_to[i]	Index of the first node to which node i is connected
last_weight_to[i]	Index of last node to which node i is connected
bias[i]	Weight vector from bias node
netinput[i]	Total input to node i
activation[i]	Activation of node i
logistic	Sigmoid activation function
weight[i][j]	Weight matrix
nunits	Number of units (nodes) in the net
ninputs	Number of input units (nodes)
noutputs	Number of output units (nodes)
error[i]	Error at node i
target[i]	Desired output of node i
delta[i]	error[i]*activation[i]*(1-activation[i])
wed[i][j]	Weight error derivatives
bed[i]	Analog wed for bias node
eta	Learning rate
momentum (a)	Reduces heavy oscillation
activation[]	Input vector

1. Calculate activation

```
.
.compute_output () {
. for (i = ninputs; i < nunits; i++) {
.   netinput[i] = bias [i];
.   for (j=first_weight_to[i]; j<last_weight_to[i]; j++) {
.    netinput[i] += activation[j]*weight[i][j];
.   }
.   activation[i] = logistic(netinput[i]);
. }
.}
.
```

2. Calculate "error"

"t" is the index for the target vector (desired output);
" `activation[i]*(1.0 - activation[i])` " is the derivative of the sigmoid activation function (the "logistic" function)
The last "for"-loop in `compute_error` is the "heart" of the backpropagation algorithm: the recursive calculation of `error` and `delta` for the hidden layers. The program iterates backwards through all nodes, starting with the last output node. For every passage through the loop, the `delta` is calculated by multiplying `error` with the derivative of the activation function. Then, `delta` goes back to the nodes (multiplied with the connection weight `weight[i][j]`). If then a specific node becomes the actual node (index "i"), the sum (`error`) is already accumulated, i.e. all the contributions of the nodes to which the actual node is projecting are already considered. `delta` is then again calculated by multiplying `error` with the derivative of the activation function:
$g' = g(1 - g)$

g' = `activation[i] * (1 - activation[1])`

```
.
.compute_error() {
. for (i = ninputs; i < nunits - noutputs; i++) {
.   error[i] = 0.0;
. }
. for (i = nunits - noutputs, t=0; i<nunits; t++, i++) {
.   error[i] = target[t] - activation[i];
. }
. for (i = nunits - 1; i >= ninputs; i--) {
.   delta[i] = error[i]*activation[i]*(1.0 - activation[i]); // (g')
.   for (j=first_weight_to[i]; j < last_weight_to[i]; j++)
.     error[j] += delta[i] * weight[i][j];
. }
.}
.
```

3. Calculating wed[i][j]

`wed[i][j]` ("weight error derivative") is `delta` of node `i` multiplied with the activation of the node to which it is connected by `weight[i][j]`. Connections from nodes with a higher level of activation contribute a bigger part to error correction (blame assignment).

```
.
.compute_wed() {
. for (i = ninputs; i < nunits; i++) {
.   for (j=first_weight_to[i]; j<last_weight_to[i]; j++) {
.     wed[i][j] += delta[i]*activation[j];
.   }
. bed[i] += delta[i];
. }
.}
.
```

4. Update weights

In this procedure the weights are changed by the algorithm. In this version of backpropagation a momentum term is used.

```
.
.change_weights () {
. for (i = ninputs; i < nunits; i++) {
.   for (j = first_weight_to[i]; j < last_weight_to[i]; j++) {
.     dweight[i][j] = eta*wed[i][j] + momentum*dweight[i][j];
.     weight[i][j] += dweight[i][j];
.     wed[i][j] = 0.0;
.   }
.   dbias[i] = eta*bed[i] + momentum*dbias[i];
```

```
.   bias[i] += dbias[i];
. bed[i] = 0.0;
. }
.}
.
```

If the `change_weights` procedure is run after every pattern presentation (i.e. after each cycle), we are dealing with the "on-line" version, if it is only run after an entire epoch, it is the "off-line" version. The "on-line" version is somehow more "natural" and less expensive in memory.

3. A historical Example: NETTalk

To illustrate the main ideas let us look at a famous example, NETTalk. NETTalk is a connectionist model that translates written English text into speech. It uses a multi-layer feedforward backpropagation network model [**Sejnowski and Rosenberg, 1987**]. The architecture is illustrated in figure 3. There is an input layer, a hidden layer, and an output layer. At the input layer the text is presented. There is a window of seven slots. This window is needed since the pronunciation of a letter depends strongly on the context in which it occurs. In each slot, one letter is encoded. For each letter of the alphabet (including space and punctuation) there is one node in each slot, which means that the input layer has 7 x 29 nodes. Input nodes are binary on/off nodes. Therefore, an input pattern, or input vector, consists of seven active nodes (all others are off). The nodes in the hidden layer have continuous activation levels. The output nodes are similar to the nodes in the hidden layer. They encode the phonemes by means of a set of phoneme features. This encoding of the phonemes in terms of phoneme features can be fed into a speech generator, which can then produce the actual sounds. For each letter presented at the center of the input window, "e" in the example shown in figure 3, the correct phoneme encoding is known. By "correct" we mean the one which has been encoded by linguists earlier[1].

The model starts with small random connection weights. It propagates each input pattern to the output layer, compares the pattern in the output layer with the correct one, and adjusts the weights according to the backpropagation learning algorithm. After presentation of many (thousands) patterns, the weights converge, i.e., the network picks up the correct pronunciation.

NETTalk is robust: i.e., superimposing random distortions on the weights, removing certain connections in the architecture, and errors in the encodings do not significantly influence the network's behavior. Moreover, it can handle - pronounce correctly - words it has not encountered before, i.e., it can generalize. The network behaves as if it had acquired the rules of English pronunciation. We say "as if" because there are no rules in the network, but its behavior is rule-like. Learning is an intrinsic property of the model. One of the most exciting properties of the model, is that at the hidden layer certain nodes start distinguishing between vowels and consonants. In other words they are on when there is a vowel at the input, otherwise

[1]In one experiment a tape recording from a child was transcribed into English text and for each letter the phoneme encoding as pronounced by the child was worked out by the linguists. In a different experiment the prescribed pronunciation was taken from a dictionary.

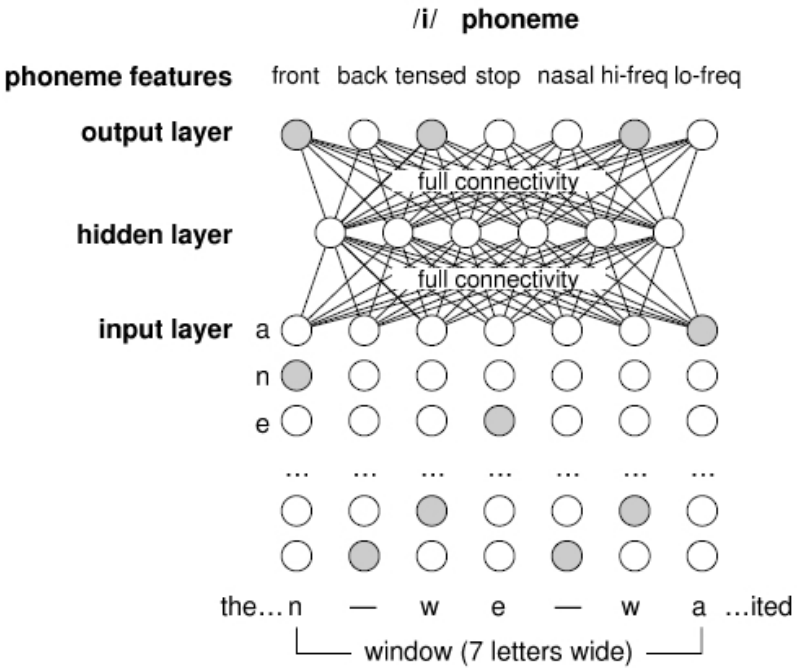


FIGURE 3. Architecture of the NETTalk model. The text shown in the window is contained in the phrase "then we waited". There are about 200 nodes in the input layer (seven slots of about 29 symbols, i.e. the letters of the alphabet, space, punctuation). The input layer is fully connected to the hidden layer (containing 80 nodes), which is in turn fully connected to the output layer (26 nodes). Encoding at the input layer is in terms of vectors of length 7 that represent a time window. Each position encodes one letter. At the output layer, the phonemes are encoded in terms of phoneme features.

they are off. As this consonant-vowel distinction has not been pre-programmed, it is called *emergent*.

NETTalk is a historic example. Current text-to-speech systems usually work in a more rule-based fashion; approaches using MLPs only are not used in practice. One important reason for this seems to be that there is an enormous amount of structure in language that is hard to extract with a "monolythic" neural network. Applying a certain amount of a prori knowledge about the structure of language certainly helps the process, which is why, in practice, combinations of methods are typically used (i.e. rule-based methods with some neural network components). This is particularly true of speech understanding systems: neural networks are never used in isolation, but for the better part in combination with HMMs (Hidden Markov Models) and rule-based components.

4. Properties of back-propagation

The backpropagation algorithm has a number of properties that make it highly attractive.

(1) *Learning, not programming.* What the network does has been learned, not programmed. Of course, ultimately, any neural network is translated into a computer program in a programming language like Java. But at the level of the neural network, the concepts are very different from traditional computer programs.
(2) *Generalization.* Back-propagation networks can generalize. For example, the NETTalk network can pronounce words that it has not yet encountered. This is an essential property of intelligent systems that have to function in the real world. It implies that not every potential situation has to be predefined in the system. Generalization in this sense means that similar inputs lead to similar outputs. This is why parity (of which XOR is an instance) is not a good problem for generalization: change one bit in the input and the output has to change maximally (e.g. from 0 to 1).
(3) *Noise and fault tolerance.* The network is noise and fault tolerant. The weights of NETTalk have been severely disturbed by adding random noise, but performance degradation was only gradual. Note that this property is not explicitly programmed into the model. It is a result of the massive parallelism, in a sense, it comes "for free" (of course, being "paid for" by the large number of nodes).
(4) *Re-learning.* The network shows fast re-learning. If the network is distorted to a particular performance level it re-learns faster than a new network starting at the same performance level. So, in spite of the low performance the network has retained something about its past.
(5) *Emergent properties*: First, the consonant-vowel distinction that the nodes at the hidden layer pick up has not been programmed into the system. Of course, whether certain nodes can pick up these distinctions depends on how the examples are encoded. Second, since the net can pronounce words that it has not encountered, it has - somehow - learned the rules of English pronunciation. It would be more correct to say that the network behaves *as if it had learned the rules of English pronunciation*: there are no rules in the network, only weights and activation levels. It is precisely these fascinating emergent properties that make the neural networks not only attractive for applications, but to researchers interested in the nature of intelligence.
(6) *Universality*: One of the great features of MLPs is that they are *universal approximators*. This has been proven by Hornik, Stinchcombe and White in 1989. More precisely, with a two-layer feed-forward network every set of discrete function values can be represented, or in a particular interval, a continuous function can be approximated to any degree of accuracy. This is still a simplification, but it is sufficient for our purposes. Note that, again, there is a difference between what can be represented and

Problem	*Trials*	*eta*	*alpha*	*r*	*Max*	*Min*	*Average*	*S.D.*
10-5-10	25	1.7	0.0	1.0	265	80	129	46

FIGURE 4. Reporting performance results

what can be learned: something that can be represented cannot necessarily be learned by the back-propagation algorithm. For example, back-propagation might get stuck in a local minimum and, except in simple cases, we don't know whether the global minimum will ever be reached.

While these properties are certainly fascinating, such networks are not without problems. On the one hand there are performance problems and on the other - from the perspective of cognitive science - there are doubts whether supervised learning schemes like backpropagation have a biological or psychological reality. In this chapter we look mainly at performance problems.

MLPs with backpropagation have been tried on very many different types of classification problems and for many years there was a big hype about these kinds of networks, on the one hand because they had been proved to be universal function approximators, and on the other because they are easy to use. Most of these experiments, however, remained in the prototype stage and were not used in everyday routine practice. We can only speculate about the reasons, but we strongly suspect that in practical situations users of information technology don't like black-boxes where they don't really know what the system actually contains. As we said before, the entire knowledge in a neural network is in the connection matrix: there are no explicit rules that would give us an idea of what the network actually "knows" and does. However, there have been a number of attempts to extract rules from neural networks, especially for Self-Organizing Maps (SOMs).

5. Performance of back-propagation

What do we mean by performance of an algorithm of this sort? We have to ask a number of questions:

- When has something been learned? Learning means fitting a model to a set of training data, such that for a given set of input patterns $\{\xi\}$, the desired output patterns $\{\xi\}$ are reproduced .
- When is the network "good"? There are actually two questions here. First, has the training set been learned, or how well has it been learned? Second, and more important, how well does the network generalize, i.e. what is the performance of the model on predicting the output on future data $\{\xi\}$. The procedure that we will look at to quantify the generalization error is called *n-fold cross-validation* (see below).
- How long does learning take?

First of all, performance should be reported in tables as the ones shown in figure 4.
eta: learning rate
alpha: momentum term
r: range for initial random weights
Max: the maximum number of epochs required to reach the learning criterion (see

below)
Min: the minimum number of epochs required to reach the learning criterion
Average: the average number of epochs required to reach the learning criterion
S.D.: the standard deviation

$$S.D. = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2} \tag{34}$$

Learning criterion
When we have binary output units we can define that an example has been learned if the correct output has been produced by the network. If we can achieve that for all the patterns, we have an error of 0. In general we need (a) a global error measure E, as we have defined it earlier and we define a critical error E_0. Then the condition for learning is $E < E_0$, (b) a maximum deviation F_0. Condition (b) states that not only should the global error not exceed a certain threshold, but the error at every output node should not exceed a certain value; all patterns should have been learned to a certain extent.

If the output units are continuous-valued, e.g. within the interval $[0 \ldots 1]$, then we might define anything < 0.4 as 0, and anything > 0.6 as 1; whatever lies between $0.4 \geq x \leq 0.6$ is considered incorrect. In this way, a certain tolerance is possible.

We also have to distinguish between performance on the training set - which is what has been reported in figure 4, for example - and on the test set. Depending on the network's ability to generalize the two performance measures can differ considerably: good performance on the training set does not automatically imply good performance on the test set. We will need a quantitative measure of generalization ability. (see section 5.3)

Let us now look at a few points concerning performance:

5.1. Convergence. *Minimizes the error function.* As in the case of adalines, this can be visualized using error surfaces (see figure 5). The metaphor used is that the system moves on the error surface to a local minimum. The error surface or error landscape is typically visualized by plotting the error as a function of two weights. Note that normally it is practically not feasible (and not needed) to calculate the entire error surface. When the algorithm runs, only the local environment of the current point in the weight space is known. This is all that's required to calculate the gradient, but we never know whether we have reached the global minimum. Error surfaces represent a visualization of some parts of the search space, i.e. the space in which the weights are optimized. Thus, we are talking about a function in weight space. Weight spaces are typically high-dimensional, so what is visualized is the error corresponding to just two weights (given a particular data set). Assume that we have the following data sets:
$\Omega_1 = \{(1.3, 1.6, 1), (1.9, 0.8, 1), (1.3, -1.0, 1), (-0.6, -1.9, 1)\}$,
$\Omega_2 = \{(-0.85, 1.7, -1), (0.2, 0.7, -1), (-1.1, 0.2, -1), (-1.5, -0.3, 1)\}$
The first two values in braces are ξ_1 and ξ_2 (the two input values), the third is the value of the function (in this case the sign function {+1,-1}). The error function depends not only on the data to be learned but also on the activation functions, except for those with linear activation functions. This is illustrated in figure 5.

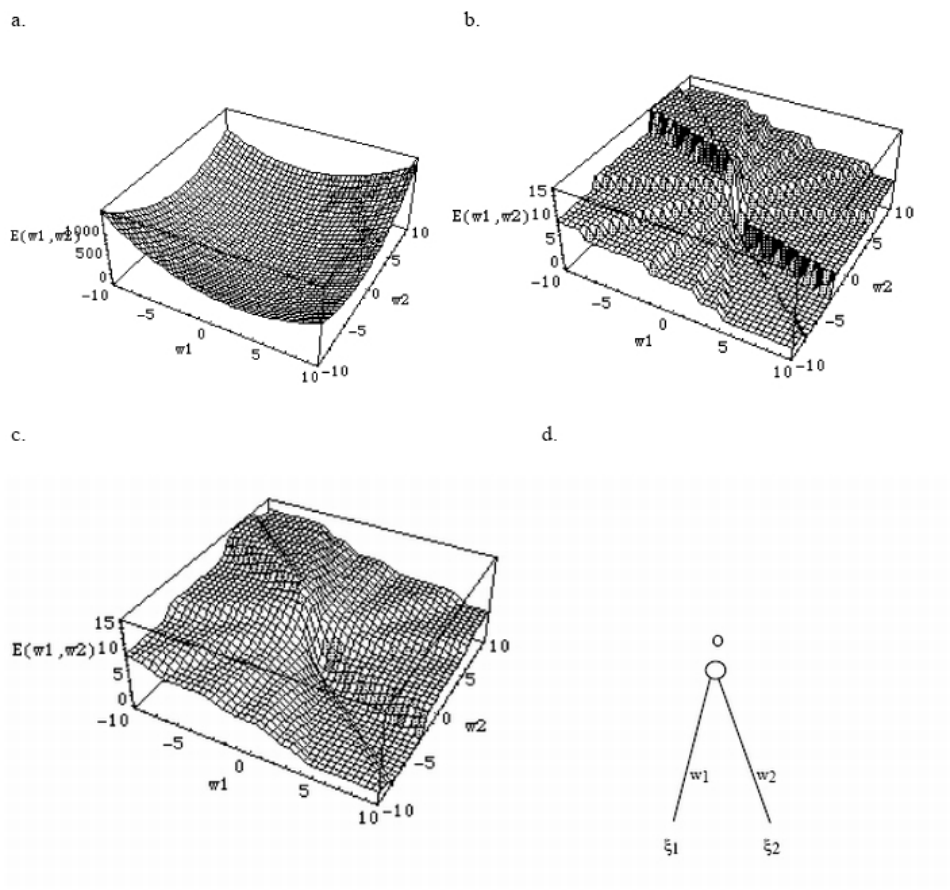


FIGURE 5. Illustration of error surfaces. The x and y axes represent the weights, the z-axis the error function. The error plots are for the perceptron shown in d. (a) linear units, (b) binary units, and (c) sigmoid units.

Local minima. One of the problems with all gradient descent algorithms is that they may get stuck in local minima. There are various ways in which these can be escaped: Noise can be introduced by "shaking the weights". Shaking the weights means that a random variable is added to the weights. Alternativly the algorithm can be run again using a different initialization of the weights. It has been argued ([**Rumelhart et al., 1986b**, **Rumelhart et al., 1986a**]) that because the space is so highdimensional (many weights) there is always a "ridge" where an escape from a local minimum is possible. Because error functions are normally only visualized with very few dimensions, one gets the impression that a back-propagation algorithm is very likely to get stuck in a local minimum. This seems not to be the case with many dimensions.

Slow convergence. Convergence rates with back-propagation are typically slow. There is a lot of literature about improvements. We will look at a number of them.

Momentum term: A momentum term is almost always added:

$$\Delta w_{ij}(t+1) = -\eta \frac{\partial E}{\partial w_{ij}} + \alpha \Delta w_{ij}(t); 0 < \alpha < 1, \text{e.g. } 0.9 \tag{35}$$

Typical value for η: 0.1. This leads to a considerable performance increase.

Adaptive parameters: It is hard to choose h and a globally once and for all. Idea: change η over time:

$$\Delta\eta = \begin{cases} +a & \text{if } \Delta E < 0 \text{ for several steps} \\ -b\eta & \Delta E > 0 \\ 0 & \text{otherwise} \end{cases}$$

There are various ways in which the learning rate can be adapted. *Newton, steepest descent, conjugate gradient, and Quasi-Newton* are all alternatives. Most textbook describe at least some of these methods.

Other optimization procedures: Many variations of the basic gradient descent method have been proposed that often yield much better performance, many of them using second derivatives. These will not be further discussed, here.

5.2. Architecture. What is the influence of the architecture on the performance of the algorithm? How do we choose the architecture such that the generalization error is minimal? How can we get quantitative measures for it?

- Number of layers
- Number of nodes in hidden layer. If this number is too small, the network will not be able to learn the training examples - its capacity will not be sufficient. If this number is too large, generalization will not be good.
- Connectivity: a priori information about the problem may be included here.

Figure 6 shows the typical development of the generalization error as a function of the size of the neural network (where size is measured in terms of number of free parameters that can be adjusted during the learning process). The data have to be separated into a training set and a test set. The training set is used to optimize the weights such that the error function is minimal. The network that minimizes the error function is then tested on data that has not been used for the training. The error on the test set is called the generalization error. If the number of nodes is successively increased, the error on the training set will get smaller. However, at a certain point, the generalization error will start to increase again: there is an optimal size of the network. If there are too many free parameters, the network will start overfitting the data set which leads to sub-optimum generalization. How can a network architecture be found such that the generalization error is minimized?

A good strategy to avoid over-fitting is to add noise to the data. The effect is that the state space is better explored and there is less danger that the learning algorithm gets stuck in a particular "corner" of the search space. Another strategy is to "grow" the network through n-fold cross-validation.

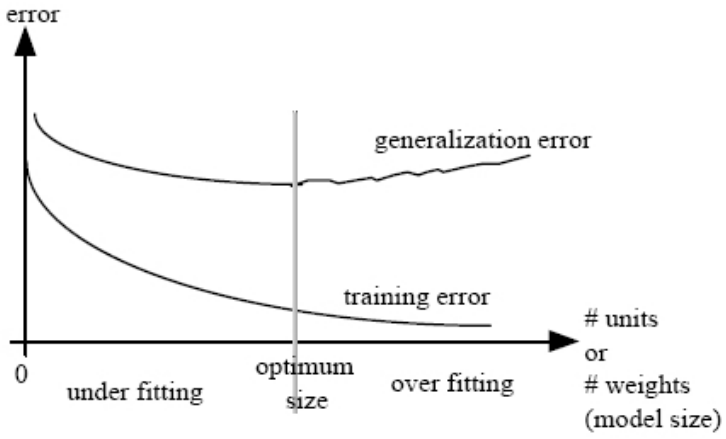


FIGURE 6. Trade-off between training error and generalization error.

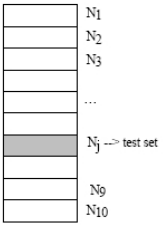


FIGURE 7. Example of partitioning of data set. One subset N_j is removed. The network $\hat{v}_j$ is trained on the remaining 9/10 of the data set. The network is trained until the error is minimized. It is then tested on the data set N_j.

5.3. N-fold cross-validation. Cross-validation is a standard statistical method. We follow [**Utans and Moody, 1991**] in its application to determining the optimal size of a neural network. Here is the procedure: Divide the set of training examples into a number of sets (e.g. 10). Remove one set (index j) from the complete set of data and use the rest to train the network such that the error in formula (36) is minimized (see figure 7).
(ξ_j, o_j): network $\hat{v}_j$
$\hat{v}_j$ is the network that minimizes the error on the entire data set minus N_j.

$$E(\underline{w}) = \frac{1}{2}\sum_{\mu,i}(\zeta_i^\mu - O_i^\mu)^2 \tag{36}$$

The network is then tested on N_j and the error is calculated again. This is the *generalization error*, CV^j. This procedure is repeated for all subsets N_j and all the errors are summed.

$$CV = \sum_j CV^j \tag{37}$$

CV means "cross-validation" error. The question now becomes what network architecture minimizes CV for a given data set. Assuming that we want to use a feed-forward network with one hidden layer, we have to determine the optimal number of nodes in the hidden layer. We simply start with one single node and go through the entire procedure described. We plot the error CV. We then add another node to the hidden layer and repeat the entire procedure. In other words, we move towards the right in figure 6. Again, we plot the value of CV. Up to a certain number of hidden nodes this value will decrease. Then it will start increasing again. This is the number of nodes that minimizes the generalization error for a given data set. CV, then, is a quantitative measure of generalization. Note that this only works if the training set and the test set are from the same underlying statistical distribution.

Let us add a few general remarks. If the network is too large, there is a danger of overfitting, and generalization will not be good. If it is too small, it will not be able to learn the data set, but it will be better at generalization. There is a relation between the size of the network (the number of free parameters, i.e. the number of nodes or the number of weights), the size of the training set, and the generalization error. This has been elaborated by [**Vapnik and Chervonenkis, 1971**]. If our network is large, then we have to use more training data to prevent overfitting. Roughly speaking, the so-called VC dimension of a learning machine is its "learning capacity".

Another way to proceed is by building the network in stages, as in Cascade-Correlation.

5.4. Cascade-Correlation. Cascade-Correlation is a supervised learning algorithm that builds its multi-layer structure during learning [**Fahlman and Lebiere, 1990**]. In this way, the network architecture does not have to be designed beforehand, but is determined "on-line". The basic principle is as follows (figure 8). "We add hidden units to the network one by one. Each new hidden unit receives a connection from each of the network's original inputs and also from every pre-existing hidden unit. The hidden unit's input weights are frozen at the time the unit is added to the net; only the output connections are trained repeatedly. Each new unit therefore adds a new one-unit "layer" to the network. This leads to the creation of very powerful higher-order feature detectors (examples of feature detectors are given below in the example of the neural network for recognition of hand-written zip codes)." ([**Fahlman and Lebiere, 1990**]).

The learning algorithm starts with no hidden units. This network is trained with the entire training set (e.g. using the delta rule, or the perceptron learning rule). If the error no longer gets smaller (as determined by a user-defined parameter), we add a new –hidden– unit to the net. The new unit is "trained" (see below), its input weights are frozen, and all the output weights are once again trained. This cycle repeats until the error is acceptably small.

To create a new hidden unit, we begin with a candidate unit that receives trainable input connections from all of the network's input units and from all pre-existing hidden units. The output of this candidate unit is not yet connected to

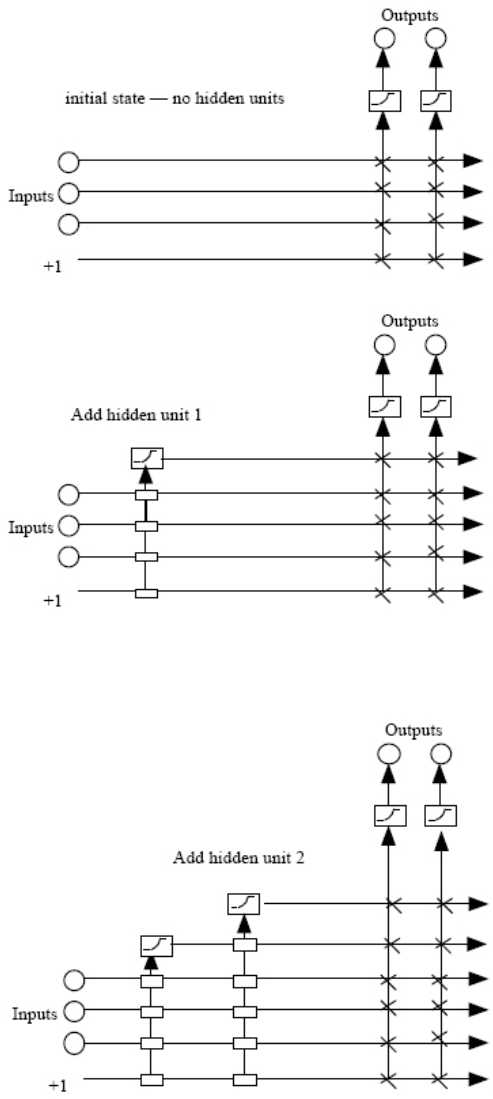


FIGURE 8. Basic principle of the Cascade architecture (after [**Fahlman and Lebiere, 1990**]). Initial state and two hidden units. The vertical lines sum all incoming activation. Boxed connections are frozen, X connections are trained repeatedly.

the active network. We run a number of passes over the examples of the training set, adjusting the candidate unit's input weights after each pass. The goal of this adjustment is to maximize S, the sum over all output units O_i of the magnitude of the correlation between V, the candidate unit's value and E_i , the residual output error observed at unit O_i . We define S as

$$S = \sum_i |\sum_p (V_p - \overline{V})(E_{p,i} - \overline{E_i})| \tag{38}$$

where i are the output units, p the training patterns, the quantities $\overline{V}$ and $\overline{E_i}$ are averaged over all patterns. S is then maximized using gradient ascent (using the derivatives of S with respect to the weights to find the direction in which to modify the weights). As a rule, if a hidden unit correlates positively with the error at a given unit, it will develop a negative connection weight to that unit, attempting to cancel some of the error. Instead of single candidate units "pools" of candidate units can also be used. There are the following advantages to cascade correlation:

- The network architecture does not have to be designed beforehand.
- Cascade correlation is fast because each unit "sees" a fixed problem and can move decisively to solve that problem.
- Cascade correlation can build deep nets (for higher-order feature detectors).
- Incremental learning is possible.
- There is no need to propagate error signals backwards through the network connections.

We have now covered the most important ways to improve convergence and generalizability. Many more are described in the literature, but they are all variations of what we have discussed here.

6. Modeling procedure

In what follows we briefly summarize how to go about when you are planning to use an MLP to solve a problem:

(1) *Can the problem be turned into one of classification?*
As we saw above, a very large class of problems can be transformed into a classification problem.

(2) *Does the problem require generalization?*
Generalization in this technical sense implies that similar inputs lead to similar outputs. Many problems in the real world have this characteristic. Boolean functions like XOR, or parity do not have this property and are therefore not suitable for neural network applications.

(3) *Is the mapping from input to output unknown?*
If the mapping from input to output is known, neural networks are normally not appropriate. However, one may still want to apply a neural network solution because of robustness considerations.

(4) *Determine training and test set. Can the data be easily acquired?*
The availability of "good" data, and a lot of data is crucial to the success of a neural network application. This is absolutely essential and must be investigated thoroughly early on.

(5) *Encoding at input. Is this straightforward? What kind of preprocessing is required?*
Finding the right level at which the neural network is to operate is essential. Very often, a considerable amount of preprocessing has to be done before the data can be applied to the input layer of the neural network. In

a digit recognition task, for example, the image may have to be normalized before it can be encoded for the input layer.

(6) *Encoding at output. Can it easily be mapped onto the required solution?* The output should be such that it can be actually used by the application. In a text-to-speech system, for example, the encoding at the output layer has to match the specifications of the speech generator. Moreover, each input letter has to be coded in terms of appropriate features which can be a considerable task.

(7) *Determine network architecture*
Experimenting with various numbers of layers and nodes at the hidden layer, using for example:
N-fold cross-validation
Cascade correlation (or other constructive algorithm)
Incorporation of a priori knowledge (constraints, see section 7.1)

(8) *Determine performance measures*
Measures pertaining to the risk of incorrect generalization are particularly relevant. In addition the test procedure and how the performance measures have been achieved (training set, test set) should be described in detail.

7. Applications and case studies

Currently, MLPs are often used in research laboratories to quickly come up with a classification system, for example, for categorizing sensory data for recognizing simple objects in the environment (which may or may not work; see type 1 and type 2 problems below), There are also a number of real-world applications, e.g. in the area of recognition of handwritten characters. We will describe this case study below, when discussing how to incorporate a priori knowledge into a neural network (sometimes called "connection engineering"). Moreover, there are interesting applications in the field of prosthetics, that we briefly describe, below.

7.1. Incorporation of a priori knowledge ("connection engineering"): handwritten zip codes. As an example, let us look at a backpropagation network that has been developed to recognize handwritten Zip codes. Roughly 10'000 digits recorded from the mail were used in training and testing the system. These digits were located on the envelopes and segmented into digits by another system, which in itself was a highly demanding task.

The network input was a 16x16 array that received a pixel image of a particular handwritten digit, scaled to a standard size. The architecture is shown in figure 9. There are three hidden layers. The first two consist of trainable feature detectors. The first hidden layer had 12 groups of units with 64 units per group. Each unit in a group had connections to a 5x5 square in the input array, with the location of each square shifting by two input pixels between neighbors in the hidden layer. All 64 units in a group had the same 25 weight values (weight sharing): they all detect the same feature. Weight sharing and the 5x5 receptive fields reduced the number of free parameters for the first hidden layer from almost 200'000 for fully connected layers to only (25+64)x12=1068. Similar arguments hold for the other layers. "Optimal brain damage" can also be applied here to further reduce the number of free parameters.

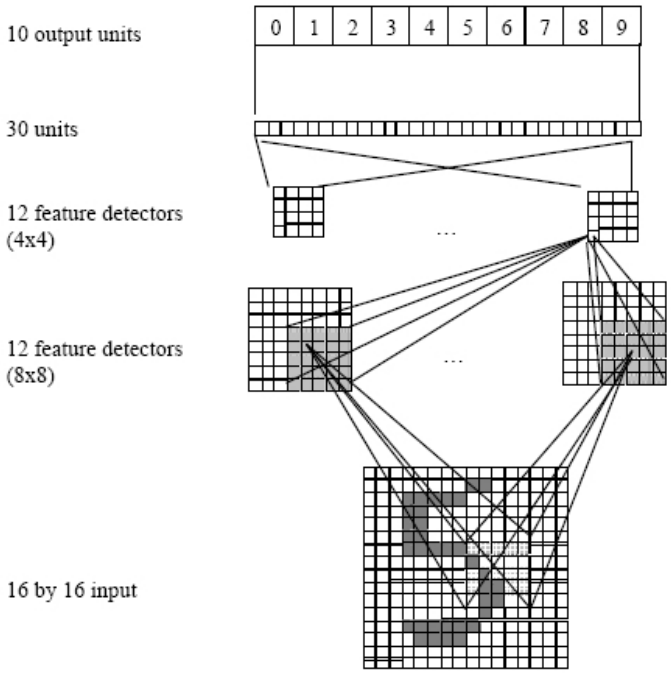


FIGURE 9. Architecture of MLP for handwritten Zip code recognition ([**Hertz et al., 1991**])

Here, the a priori knowledge that has been included is: recognition works by successive integration of local features. Features are the same whether they are in the lower left or upper right corner. Thus, weight sharing and local projective fields can be used. Once the features are no longer local, this method does not work any more. An example of a non-local feature is connectivity: are we dealing with one single object?
Other approaches using various types of neural networks for hand-written digit recognition have also been formulated (e.g., [**Pfister et al., 2000**, **Cho, 1997**]).

In the late 1980s and early 1990s there was a lot of enthusiasm about MLPs and their potential because they had been proven to be universal function approximators. They were tried on virtually any problem that could be mapped onto one of classification. We give a very brief description of how developers imagined these applications could be handled using neural networks.
Let us look at automatic driving, stock market prediction, and distinguishing metal cylinders from rocks.

ALVINN

ALVINN, the autonomous land vehicle in a neural network (e.g. [**Pomerleau, 1993**]) works by classifying camera images. The categories are, as pointed out earlier, the steering angle. The system is trained by taking a lot of camera images from road scenes for which the steering angle is provided. More recent versions of ALVINN provide networks for several road types (4-lane highway, standard highway,

etc.). It first tests for which one of these there is most evidence. ALVINN has successfully navigated over large distances. Neural networks are a good solution to this problem because it is very hard to determine the steering angle by logical analysis of the camera image. In other words, the mapping from images to steering angle is not known. Recent versions of automated driving systems, developed by the same researcher, Dean Pomerleau of Carnegie-Mellon University, do not use neural networks. The more traditional methods from computer vision and multi-sensory fusion according to Pomerleau seem to work best.

Stock market prediction

Again, this is a problem where little is known a priori about the relation between the past events, economic indicators, and development of stock prices Various methods are possible. One is to take the past 10 values and trying to predict the 11th. This would be the purely statistical approach that does not incorporate a priori knowledge of economic theory. It is also possible to include stock market indicators into a neural network approach. Typically combinations of methods are used.

Distinguishing between metal cylinders and rocks

A famous and frequently quoted example of an application of backpropagation is also the network capable of distinguishing between metal cylinders (mines) and rocks based on sonar signals [**Gorman et al., 1988a**, **Gorman et al., 1988b**]. This is another instance where a direct analysis of the signals was not successful: the mapping of the signals to the categories is not known. Whether this application is routinely used is actually unknown because it is a military secret.

Summary: although neural networks are theoretically arbitrarily powerful, other methods that can more easily incorporate domain knowledge or extract "hidden structure" (e.g., HMMs) are often preferred. A recent application using a feed-forward neural network that seems to work very well is a prosthetic hand.

7.2. Mutual adaptation between a prosthetic hand and patient. Figure 10 shows the robotic hand developed by Hiroshi Yokoi and Alejandro Hernandez at the Uniersity of Tokyo (see [**Gomez et al., 2005**]). The hand can also be used as a prosthetic hand. It can be attached to patients non-invasively by means of EMG electrodes. EMG stands for ElectroMyoGram. EMG electrodes can pick up the signals on the surface of the skin when muscles are innervated. Patients with an amputated hand can still produce these signals which can then be used to control the movements of the hand. One of the main problems is that the kinds of signals patients are capable of producing are highly individual, depending on exactly where the amputation was made and when.Yokoi developed an interactive training procedure whereby patients learn to produce signals that can be nicely mapped onto movements of the hand. Thus, the preconditions for applying an MLP approach are largely fulfilled; the input data are the EMG signals, the desired output particular movements such as grasping with all finger or movement of the thumb.

Patients have visual feedback from the hand, i.e. they can see the effect of their producing muscle innervations that can be picked up by the EMG electrodes. Tactile feedback in the form of light electrical stimulation is also being tested. The idea here is that with a normal hand, we get a lot of tactile feedback which is essential for learning. Even though the tactile feedback is of a very different nature, it still seems to improve learning.

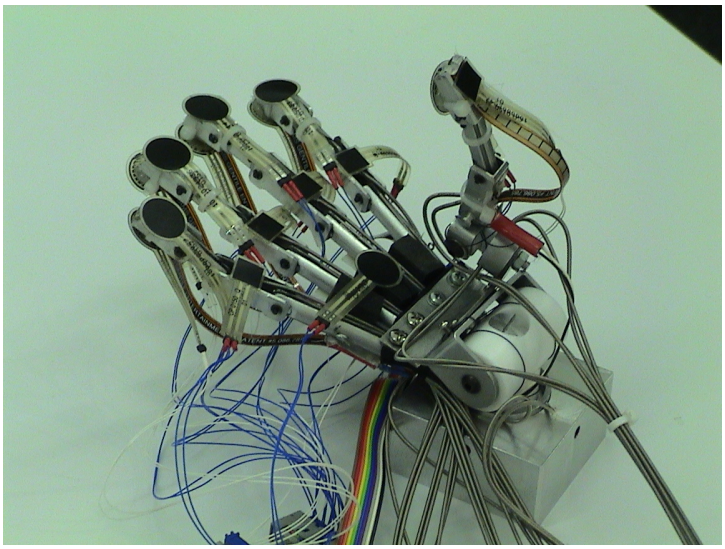


FIGURE 10. Robotic Hand

This method of attaching a prosthetic hand non-invasively via EMG electrodes is currently being tested on patients in Japan. In addition to the training procedure with the MLP, fMRI studies are conducted to see how the brain activation changes as patients improve their skills at using the hand.

8. Distinguishing cylinders from walls: a case study on embodiment

Stefano Nolfi of the Italian center for scientific research in Rome made a series of highly instructive experiments with Khepera robots. The details can be found in the book "Understanding intelligence" ([**Pfeifer and Scheier, 1999**], p. 388 ff.).

The task of the robot was to distinguish cylinders from walls. While this task, from the perspective of the human observer, may seem trivial and uninteresting, from the situated perspective of the robot, it may actually be quite demanding. They used an MLP for the training. Because for MLPs to work properly, a lot of data (input–desired output pairs) are required. They collected sensory data at 20 different distances and 180 orientations. The connectivity was as follows: 6 input nodes (one for each IR sensor) and 1 output node. For the hidden layer, they allocated zero, four, or eight nodes. The results can be seen in [**Pfeifer and Scheier, 1999**], p. 390.

What is surprising, at least at first sight, is the fact that there are large regions where the distinction could not be learned (the white regions). While it is obvious that on the left and the right there are white regions, simply because there are no sonsors in these orientations, it is less obvious why there would be white regions near the wall. Apparently, from a situated perspective, the two, especially from nearby, are presumably not sufficiently different and they do not constitute a learnable function. In other words, the data in the white regions are of "Type 2" (see [**Clark and Thornton, 1997**]). By increasing the VC dimension of the

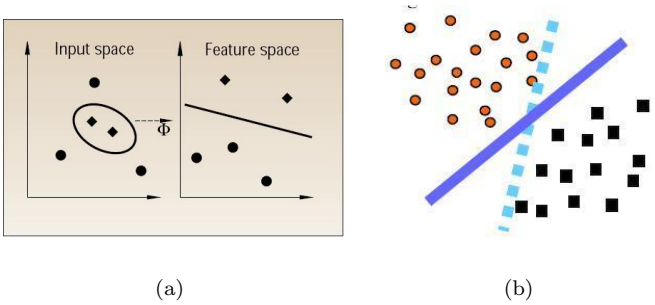
network, or stated differently, by amplifying the level of computation, the improvements are only minimal, as can be seen in the experiments with four and eight nodes (adding nodes to the hidden layer increases the number of free parameters of the learning machine and thus its VC dimension). Thus, we can see that learning is not only a computational problem. In order to learn the categories in case of a type 2 problem, either additional information is required, or the data have to be improved. One way of generating additional data is to engage in a process of sensory-motor coordination. Nolfi and his colleagues, in a set of experiments where they used artificial evolution to determine the weights, showed that indeed the fittest agents, i.e. the ones that could best distinguish between cylinders and walls, are the ones that do not sit still in front of the object, but that engage in a sensory-motor coordination. This prediction is consistent with the principle of sensory-motor coordination (see [**Pfeifer and Scheier, 1999**], chapter 12, for more detail).

Summary and conclusions. MLPs are probably the best investigated species of neural networks. With the advent of the mathematical proof of their universality, there was a real hype and people have tried to apply them everywhere, as already mentioned earlier. As always, the sense of reality "kicks in" when the first hype is over. While in some areas, MLPs have proved extremely useful (recognizing hand-written characters; non-invasive attachment of robotic hand; experiments in research laboratories), in others, traditional methods from statistics or mathematical modeling seem to be superior (e.g. automatic driving). In yet other fields, typically combinations of methods are used (speech understanding, stock market prediction).

Natural brains have evolved as part of organisms that had to be adaptive in order to survive in the real world. Thus, the main role of brains is to make organisms adaptive. Wherever there is a real need for systems to be adaptive, neural networks will be useful and good tools. Where adaptivity is not a core issue, typically other methods are to be preferred. Because in biological systems, supervised learning methods don't seem to exist (at least not in the sense defined here), it may indeed be the case that neural networks are in fact most suited for applications where adaptivity is required. While some applications requiring adaptivity can be dealt with in terms of MLPs, often non-supervised networks will be required (see the following chapters).

9. Other supervised networks

Support Vector Machines belong to the type of supervised learning methods used for classification and regression. The underlying idea consists in constructing a hyperplane that best separates classes. If this is not feasible in the input space, the goal is achieved in a higher dimensional space, called feature space, reached by mapping the input vector through a nonlinear function. This mapping has the property that will linearly separate the data sets in the feature space. The hyperplane in the feature space will maximally separate the data set classes. See Figure 11(a) for a pictorial description of this concept. The method is founded based on statistical learning theory and is successfully used in a number of applications like particle identification, face detection, text categorization. Technically, it is mathematically intuitive, reproducible, does not have problems of local minima as the optimization function is convex, and is robust to new classification data tasks.



(a) (b)

FIGURE 11. Data sets and their separation. (a) Map the training data nonlinearly into a higher dimensional feature space and construct a hyperplane with maximum margins there [**Hearst, 1998**], (b) Two potential hyperplane that separate data sets.

Let it be the data points x_i and x_j that must be binary classified into two classes $y_i = 1$ and $y_j = -1$, respectively. To this aim, let us consider the classification function $f = sign(wx - b)$ where w establishes the orientation of the hyperplane in the feature space, and b represents the offset of the hyperplane from the origin. For a more robust classification and better generalization of future data, it is pretty intuitive that the correspondent hyperplane should maximally apart the data sets, which is the case for the continuous line in Figure 11(b). The optimal hyperplane is a perpendicular bisector of the shortest line that connects the convex hulls of each data set(the dotted contour in Figure 12).

The problem of finding the optimal hyperplane transforms into that of finding the closest points each of which belonging to a different convex hull. This will give the hyperplane that intersects these points, and the equation of the hyperplane that optimally separates the data sets yields as the hyperplane that bisects the former. Determining the parameters of the hyperplane's equation thus implies finding the minimum distance between the convex hulls which can be solved by many algorithms that deal with quadratic problems.

Another approach in the data sets partitioning is to maximize the distance between two hyperplanes that are parallel and which keep a whole data set clustered in a single semi-hyperplane. These two hyperplanes are each depicted with a thin line in Figure 12, for the case of a two dimensional data. The data points that lie on these hyperplanes are called support vectors and they carry the most relevant information about the classification issue.

Since the equation of the plane that bisects the convex hulls of the data sets is $wx - b = 0$, the equations of the two supporting semiplanes will be $wx - b > 0$ and $wx - b < 0$. Rewriting these inequalities as $wx - b > 1$ and $wx - b < -1$ will not deprive of generality due to the fact that the decision function is invariant to positive rescaling. Thus we can compute the distance(or the margin) between the two supporting planes according to the following simple arithmetic:

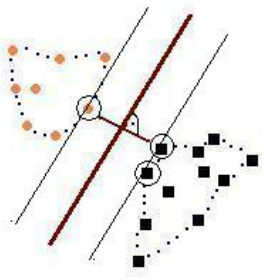


FIGURE 12. Data sets separated by a hyperplane or supported by two parallel hyperplanes

$$\begin{gathered} wx_i - b = 1 \\ wx_j - b = -1 \\ \Rightarrow w(x_i - x_j) = 2 \\ \Rightarrow \left(\frac{w}{||w||}\right)(x_i - x_j) = \frac{2}{||w||} \end{gathered}$$

Maximizing the distance $\frac{2}{||w||}$ means minimizing the value $||w||$:

$$min_{w,b}\left(\frac{||w||^2}{2}\right), \tag{39}$$

under the imposed conditions that

$$\begin{gathered} wx_i - b > 1, \text{ for the class } y_i \text{ and} \\ wx_j - b < -1, \text{ for the class } y_j. \end{gathered}$$

The Lagrangian dual of the previous formulation yields the following quadratic programming(QP) expression:

$$min_\alpha\left(\frac{1}{2}\right) * \sum\sum y_i y_j \alpha_i \alpha_j x_i x_j - \sum \alpha_i \tag{40}$$

such that

$$\sum y_i \alpha_i = 0 \text{ and } \alpha_i \geq 0$$

for which standard solving algorithms exist. As in the other alternative case, this QP problem gives at optimum the values of $w = \sum y_i \alpha_i x_i$ and b, the coefficients of the target hyperplane.

The case illustrated above corresponds to linear separable data. This assumption cannot hold for all types of data distribution. In most of the cases, the low number of the available variables on which the data depends places it heterogeneously within the data space (Figure 13, left image).

For this case, the solution is to restrict the influence of those points that intrude into a class they don't belong to. This is achieved by relaxing the convex hull for the data class of these points as shown in Figure 13, right image.

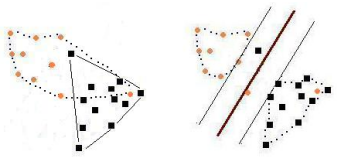


FIGURE 13. Left image: linearly inseparable data. Right image: linear delimitation by relaxing the convex hulls

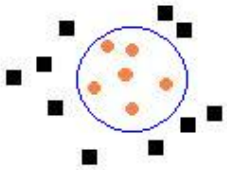


FIGURE 14. Nonlinear discriminant

The quadratic problem for the classification of the linearly inseparable data resorts to a trade-off between maximizing the distance between the reduced convex hulls and simultaneously to minimize the error attributable to the points left apart from the belonging class. These considerations are formalized as follows:

$$min_{w,b,z_i,z_j}\left(\left(\frac{||w||^2}{2}\right)+C\sum z_i+D\sum z_j\right), \tag{41}$$

imposing the conditions that

$$wx_i-b+z_i>1 \text{ , for class } y_i \text{ and}$$
$$wx_j-b+z_j<-1 \text{ , for the class } y_j,$$

where $z_i>0, z_j>0$.
In these equations, z_i and z_j, called slack variables, are representatives for the error contribution of each data point isolated from its corresponding class y_i and y_j respectively. When summing them up, a weighted penalty(C and D) is assigned to each term.

The idea of relaxing the influence of the data points reflects on the dual QP problem formulation by introducing a small change within:

$$min_{\alpha}\left(\frac{1}{2}\right)*\sum\sum y_iy_j\alpha_i\alpha_jx_ix_j-\sum\alpha_i, \tag{42}$$

such that $\sum y_i\alpha_i=0$ and $C\geq\alpha_i\geq 0$.

Searching for a linear discriminant in the original space proves to be ineffective in terms of the error quantization, nonetheless, when the data is scattered as shown in Figure 14.

Through a mapping in some other space, called the feature space, the aim for the data there is to be classified linearly. By adding attributes that are nonlinear functions of the original data, nonlinearity characteristic is preserved in the initial space, whereas the feature space can be set for further linear analysis.

For instance, for a two dimensional initial space with attributes $[r, s]$, it is possible to find a mapping into a five dimensional feature space $[r, s, rs, s^2, r^2]$:
$x = [r, s]$ with $wx = w_1 r + w_2 s, \rightarrow \theta(x) = [r, s, rs, s^2, r^2]$ with $w\theta(x) = w_1 r + w_2 s + w_3 rs + w_4 s^2 + w_5 r^2$
The classification function can thus be chosen to be:

$$f(x) = sign(w\theta(x) - b) = sign(w_1 r + w_2 s + w_3 rs + w_4 s^2 + w_5 r^2 - b)$$

As it can be seen, $f(x)$ is linear in the feature space and nonlinear in the initial space.

A problem that occurs in this technique is that for high dimensional spaces, the computability of θ becomes awkward and inefficient due to the number of operations needed to process it. In the equation below, we would have to compute twice the mapping of θ and then the actual dot product between their expressions.

$$min_\alpha \left(\frac{1}{2}\right) * \sum\sum y_i y_j \alpha_i \alpha_j \theta(x_i)\theta(x_j) - \sum \alpha_i \tag{43}$$

such that

$$\sum y_i \alpha_i = 0 \text{ and } C \geq \alpha_i \geq 0$$

But due to the fact that at the core of the optimization problem of the SVMs lies a dot product operation between the θ mappings of the classes (as seen in the equation 43), the issue is bypassed via Mercer's Theorem which states that there are certain mappings θ and kernel functions K, such that $\theta(x_1)\theta(x_2) = K(x_1, x_2)$, given any two points x_1 and x_2; by knowing K, the computation of θ mapping is ignored, and thus the dot product is simply replaced by this kernel function.

Here is a short list of some standard kernel functions associated with specific quadratic mappings:

θ	$K(x_1, x_2)$
d-degree polynomial	$(x_1 x_2 + 1)^d$
RBF	$exp\left(-\frac{\lvert x_1 - x_2\rvert^2}{2\sigma}\right)$
two layer NN	$sigmoid(\eta(x_1 x_2 + 1) + k)$

Therefore, the dual QP problem will alter according to the expression:

$$min_\alpha \left(\frac{1}{2}\right) * \sum\sum y_i y_j \alpha_i \alpha_j K(x_i, x_j) - \sum \alpha_i \tag{44}$$

such that

$$\sum y_i \alpha_i = 0 \text{ and } C \geq \alpha_i \geq 0$$

The kernel function has the merit of avoiding the necessity of computing functions in a high dimensional space - its standard form controls the dimensionality of the computations this time, is still expressed in terms of the input data points and efficiently masks the mapping to the feature space where the linear discriminator is more likely to be found. The transformations that comes along do not change

the nature of the optimization problem, the function still preserves a convex shape that guarantees a global minimum.

The SVM algorithm can be summarized in the following four steps:

(1) Select parameter C that restraints the influence of one data point. Select the type of the kernel and the associated parameters.
(2) Solve the dual QP problem or an alternative formulation.
(3) Determine parameter b by means of the hyperplane equation.
(4) Classify a new data point x point by passing it to the decision function: $f(x) = sign(\sum y_i \alpha_i K(x_i, x) - b)$.

CHAPTER 5

Recurrent networks

So far we have been studying networks that only have forward connections and thus there have been no loops. Networks with loops - recurrent connections is the technical term used in the literature - have an internal dynamics. The networks that we have introduced so far have only retained information about the past in terms of their weights, which have been changed according to the learning rules. Recurrent networks also retain information about the past in terms of activation levels: activation is preserved for a certain amount of time: it is "passed back" through the recurrent connections. This leads to highly interesting behaviors, behaviors that can be characterized in terms of dynamical systems. We will therefore briefly introduce some elementary concepts from the area of dynamical systems.

The most important type of network that we will discuss in this chapter are the Hopfield nets. Because of their properties, they have inspired psychologist, biologists, and physicists alike. Associative memories - also called content-addressable memories - can be realized with Hopfield nets, but also models of physical systems like spin-glasses can be modeled using Hopfield networks. Content-addressable memories are capable of reconstructing patterns even if only a small portion of the entire pattern is available.

We will first discuss Hopfield nets with all their implications and then look at recurrent neural networks, such as recurrent backpropagation, CTRNNs – Continuous Time Recurrent Neural Networks, and echo-state networks.

1. Basic concepts of associative memory - Hopfield nets

Associative memory is one of the very fundamental problems of the field of neural networks. The basic problem of associative memory can be formulated as follows:
"Store a set of p patterns $\underline{\xi^\mu}$ in such a way that when presented with a new pattern ζ , the network responds by producing whichever one of the stored patterns most closely resembles ζ." ([**Hertz et al., 1991**], p. 11).

The set of patterns is given by $\{\underline{\xi^1}, \underline{\xi^2}, \ldots, \underline{\xi^p}\}$, the nodes in the network are labeled $1, 2, \ldots, N$. A pattern of activation in Hopfield nets always includes all the nodes. The patterns are binary, consisting of values $\{0, 1\}$ or alternatively $\{-1, +1\}$. The former can be translated into the latter as follows. Let n_i be values from the set $\{0, 1\}$. These values can be transformed into $\{-1, +1\}$ simply by $S_i = 2n_i - 1$. The symbol S_i always designates units that assume values $\{-1, +1\}$.
Remember, how is a Hopfield net constructed? First, we have to define the node characteristics. The nodes are binary threshold - in this chapter the units will have the values $\{-1, +1\}$. The connectivity matrix is as follows:

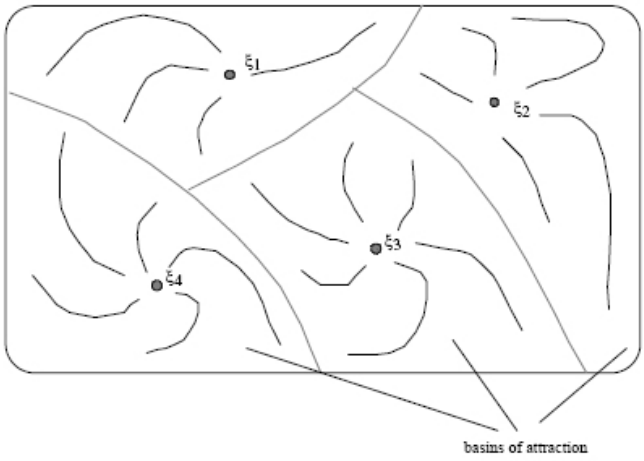


FIGURE 1. Basins of attraction. Within the basins of attraction the network dynamics will move the state towards the stored patterns, the attractors.

(45)

$*$	1	2	3	4	5
1	0	w_{12}	w_{13}	w_{14}	w_{15}
2	w_{21}	0	w_{23}	w_{24}	w_{25}
3	w_{31}	w_{32}	0	w_{34}	w_{35}
4	w_{41}	w_{42}	w_{43}	0	w_{45}
5	w_{51}	w_{52}	w_{53}	w_{54}	0

The weights are symmetric, i.e. $w_{ij} = w_{ji}$ and the nodes are not connected to themselves (0's in the diagonal).
Assume now, that ξ^{μ_0} is a stored vector. Assume that $\zeta = \xi^{\mu_0} + \Delta$, where Δ is the deviation from the stored pattern. If through the dynamics of the network (see below) ζ moves towards ξ^{μ_0}, ξ^{μ_0} is called an *attractor* and the network is called *content-addressable*. It is called content-addressable because a specific item in memory is retrieved not by its storage location or address but by recalling content related to it. In other words, the network is capable of reconstructing the complete information of a pattern from a part of it or from a partially distorted pattern. A term often used in the literature for this process is *pattern completion*.

1.1. Non-linear dynamical systems and neural networks. There is a vast literature on dynamical systems, and although at a high level there is general agreement on the basic concepts, a closer look reveals that there is still a considerable diversity of ideas. We will use the terms *dynamical systems, chaos, nonlinear dynamics,* and *complex systems* synonymously to designate this broad research field, although there are appreciable differences implied by each of these terms. Our purpose here is to provide a very short, informal overview of the basic notions that we need for the . Although we do not employ the actual mathematical theory, we will make use of the concepts from dynamical systems theory because they provide a highly intuitive set of metaphors for the behavior of neural networks.

A dynamical system is, generally speaking, a system that changes according to certain laws: examples are economical systems, the weather, a swinging pendulum, a swarm of insects, an artificial neural network (e.g. a Hopfield net, an echo-state networks, CTRNN – Continuous Time Recurrent Neural Networks, and a brain. Dynamical systems can be modeled using differential equations (or their discrete analogs, difference equations). In neural networks, the laws are the update rules for the activation (e.g. the Hopfield dynamics, see below). The mathematical theory of dynamical systems investigates how the variables in these equations change over time. However, to keep matters simple, we will use differential equations only rarely in this course, but simply state the update rules for the activation levels: $a(t+1) = \text{const}.a(t) + ...$

One of the implications of nonlinearity is that we can no longer, as we can with linear systems, decompose the systems into subsystems (e.g. sub-networks), solve each subsystem individually, and then simply reassemble them to give the complete solution. In real life, this principle fails miserably: if you listen to two of your favorite songs at the same time, you dont double your pleasure! (We owe this example to [**Strogatz, 1994**]) Another important property of nonlinear systems is their *sensitivity to initial conditions*: if the same system is run twice using very similar initial states, after a short period of time, they may be in completely different states. This is also in contrast to linear systems, in which two systems started similarly will behave similarly. The weather is a famous example of a nonlinear system–small changes may have enormous effects–which is what makes weather forecasting so hard. The *phase space* of a system is the space of all possible values of its important variables. If we consider the activation levels of the nodes in a network with n nodes, the phase space has n dimensions. For a four-legged robot, for example, we could choose the joint angles as important variables and characterize its movement by the way the angles change over time. If there are two joints per leg, this yields an eight-dimensional phase space: each point in phase space represents a set of values for all eight joints. Neighboring points in phase space represent similar values of the joint angles. Thus we can say that these changes are analogous to the way the point in phase space (the values of all joint angles at a particular moment) moves over time. The path of this point in phase space, i.e., the values of all these joint angles over time, is called the *trajectory* of the system. An *attractor state* is a preferred state in phase space toward which the system will spontaneously move if it is within its *basin of attraction*. There are four types of attractors: point, periodic, quasi-periodic, and chaotic. It is important to realize that the attractors will always depend on the laws of the change (in our case, the update rules), and on the initial conditions.

If the activation levels of a neural network, after a short period of time, will more or less repeat, which means that the trajectory will return to the same location as before, this cyclic behavior is known as a *periodic attractor* (we also talk about a *limit cycle*). If the values of the variables are continuous, which implies that they will never exactly repeat it is called a *quasi-periodic* attractor. In Hopfield nets with discrete values of the activation levels, we often have truly periodic attractors. *Point attractors* or *fixed points* are trajectories which lead the network to dwell for an appreciable period of time on a single state. These are generally the least

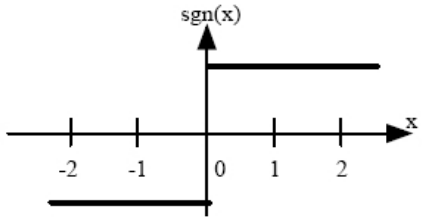


FIGURE 2. The function $sgn(x)$.

sensitive to update procedures and are rather insensitive to initial conditions (i.e. they normally have basins of attraction of "good" size). Finally, if the trajectory moves within a bounded region in the phase space but is unpredictable, this region is called a *chaotic attractor*. Systems tend to fall into one of their attractors over time: the sum of all of the trajectories that lead into an attractor is known as the *basin of attraction*. While the notion of an attractor is powerful and has intuitive appeal, it is clear that transitions between attractor states are equally important, e.g., for generating sequences of behavior (but in this course, we do not study networks that automatically change from one attractor state to another).

We should never forget the fifth basic which is the embedding of the neural network in the real world, i.e. it will often be coupled to a physical system. Because this physical system, e.g. a robot, also has its own characteristic frequencies, the entire system will have attractor states that are the result of the synergistic interactions of the two. This phenomenon is known as *mutual entrainment*: the resulting frequency will represent a "compromise" between the systems involved. For those who would like to know more about the mathematical foundations of dynamical systems we recommend [**Strogatz, 1994**], for those interested in its application to cognition, [**Port and Van Gelder, 1995**] and [**Beer, 2003**]. The book by [**Amit, 1989**] shows how attractor neural networks can be applied to modeling brain function.

1.2. Propagation rule - dynamics. The propagation rule, or in other words, the network dynamics (called the Hopfield dynamics), the law that accounts for change, is defined as follows:

$$S_i = sgn\{\sum_j w_{ij} S_j - \Theta_i\} \tag{46}$$

$$\text{where } sgn(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases} \tag{47}$$

Graphically, the *sgn* function is shown in figure 2.
In this chapter we set $\Theta = 0$. Thus, we have

$$S_i = sgn\{\sum_j w_{ij} S_j\} \tag{48}$$

A better way of writing this would be

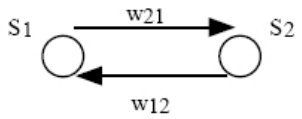


FIGURE 3. The simplest possible Hopfield net. It is used to illustrate the fact that synchronous update can lead to oscillations whereas asynchronous update leads to stable behavior.

		synchronous update				asynchronous update	
	state	S1	S2	S1	S2	S1	S2
t							
1		-1	+1	+1	+1	+1	+1
2		-1	+1	-1	-1	-1	+1
3		stable		+1	+1	-1	+1
4				-1	-1	stable	
5				oscillates			

FIGURE 4. Temporal development for the network shown in figure 3.

$$S_i^{t+1} = sgn(\sum_j w_{ij} S_j^t) \tag{49}$$

to clearly indicate the temporal structure.

The updating can be carried out essentially in two ways (with variations), *synchronously* and *asynchronously*. Synchronous update requires a central clock and can lead to oscillations. To illustrate this point let us look at the simplest possible Hopfield net (figure 3).

Figure 4 shows the development of the activation over time for weight values $w_{12} = w_{21} = -\frac{1}{2}$.

The asynchronous update rule is to be preferred because it is more natural for brains and leads to more stable behavior (although brain activity can synchronize – and synchronization processes are very important – they are not centrally clocked). Asynchronous updating can be done in (at least) two ways:

- One unit is selected randomly for updating using $S_i^{t+1} = sgn(\sum_j w_{ij} S_j^t)$
- Each unit updates itself with a certain probability.

In the asynchronous case the updating procedure is run until a stable state has been reached.

1.3. Single pattern. Let us now assume that we want to store one single pattern $\underline{\xi} = (\xi_1, \xi_2, \ldots, \xi_n), e.g. (1, -1, -1, 1, -1)$. What does it mean to "store a pattern" in the network? It means that if the network is in this state, it remains in this state. Moreover, the pattern should be an attractor in the sense defined earlier, so that pattern completion can take place. The stability condition is simply:

$$S_i^{t+1} = S_i^t \tag{50}$$

or

$$sgn(\sum_j w_{ij}\xi_j) = \xi_i, \forall i(\text{i.e. for all nodes}) \tag{51}$$

ξ_j is the to-be-stored pattern. How do we have to choose the weights such that condition (51) is fulfilled? We will show that this is the case if the weights are chosen as follows:

$$w_{ij} \propto \xi_i\xi_j$$

If we take $\frac{1}{N}$ as the proportionality factor we get

$$w_{ij} = \frac{1}{N}\xi_i\xi_j \tag{52}$$

Substituting (52) into the equation for the dynamics (49) we get
$S_i^{t+1} = sgn(\sum_j w_{ij}\xi_j) = sgn(\frac{1}{N}(\sum_j \xi_i\xi_j\xi_j)) = \xi_i$
which corresponds to the stability condition (51). Note that we always have $\xi_j\xi_j = 1$. Moreover, if we start the network in a particular state S_i , the network will converge to pattern ξ_i if more than half the bits of S_i correspond to ξ_i . This can be seen if we calculate the net input h_i

$$h_i = \sum_j w_{ij}S_j = \frac{1}{N}\sum_j \xi_i\xi_j S_j = \frac{1}{N}\xi_i \sum_j \xi_j S_j \tag{53}$$

If more than half of the $\xi_j S_j$ are positive ($\xi_j = S_j$, the sum will be positive and the sign of S_j will not change.
Let us look at an example:

$\xi_i = (1, -1, -1, 1, 1)$ stored pattern
$S_i^t = (1, -1, 1, -1, 1)$ current state of network
$\xi_j S_j^t = (1, 1, -1, -1, 1)$
$\sum_j \xi_j S_j^t = 1 \rightarrow sgn(h_i) = sgn(\xi_i)$
Because $sgn(h_i) = S_j^{t+1}$ we get
$S_i^{t+1} = (1, -1, -1, 1, 1)$
$\xi_j S_j^{t+1} = (1, 1, 1, 1, 1)$

and thus $S_j^{t+1} = S_j^t$, which means that it is a stable state. In other words, ξ_i is an attractor. Actually in this simple case there are two attractors, ξ_i and $-\xi_i$. The latter is also called the reversed state (or inverse attractor) - an instance of the so-called spurious states. Spurious states are attractors in the network that do not correspond to stored patterns. We will discuss spurious states later on. Figure 5 shows the stored pattern and the spurious attractor.

1.4. Several patterns. If we want to store p patterns we simply use a superposition of the 1-pattern case:

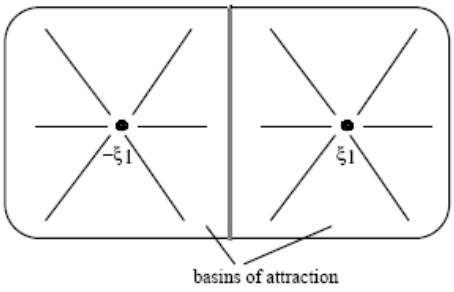


FIGURE 5. Schematic state space. Pattern including reverse state. There are two basins of attraction.

$$w_{ij} = \frac{1}{N}\sum_{\mu=1}^{p} \xi_i^\mu \xi_j^\mu \tag{54}$$

This is sometimes called Hebb's rule. This is because of an analogy of (54) with Hebb's suggestion that the synapses are strengthened if there is simultaneous activity in two nodes, post-synaptic and presynaptic (it is a variation of Hebb's rule where the pattern ξ^μ strengthens the weights between nodes i and j of $\xi_i^\mu = \xi_j^\mu$, and weakens them if they differ). However, as stated here, it is a bit unnatural, since the weights are calculated directly, not through a learning process. An associative memory with binary units and asynchronous updating, using rule (54) to calculate the weights and (49) as the dynamics, is called a *Hopfield model*.
Let us again look at the stability condition for a particular pattern ν:

$$sgn(h_i^\nu) = \xi_i^\nu, \forall i$$

where

$$h_i^\nu = \frac{1}{N}\sum_{j=1}^{N} w_{ij}\xi_j^\nu = \frac{1}{N}\sum_{j=1}^{N}\sum_{\mu=1}^{p} \xi_i^\mu \xi_j^\mu \xi_j^\nu \tag{55}$$

If we now separate out the term ξ_j^ν we get

$$h_i^\nu = \xi_j^\nu + \frac{1}{N}\sum_{j=1}^{N}\sum_{\mu=1,\mu\neq\nu}^{p} \xi_i^\mu \xi_j^\mu \xi_j^\nu \tag{56}$$

The second term in (56) is called the *crosstalk term*. If it is zero, we have stability. But even if it is not zero, we can still have stability if its magnitude is smaller than 1, in which case it cannot change the sign of h_i^ν. It turns out that if this is the case, and the initial state of the network is near one of the stored patters (in Hamming distance), the network moves towards ξ_i^ν, i.e. ξ_i^ν is an attractor. The more patterns that are stored, the lower the chances that the crosstalk term is sufficiently small.

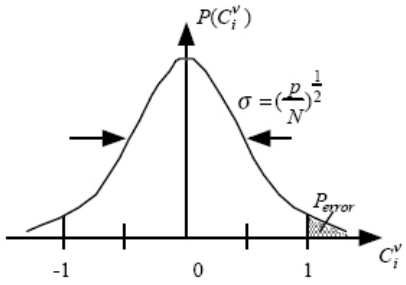


FIGURE 6. The distribution of values for the quantity C_i^ν (57). For p random patterns and N units this is a Gaussian with variance $\sigma^2 = \frac{p}{N}$. The shaded area is P_{error} , the probability of error per bit.

1.5. Storage capacity. The number of symbols that can be stored in n bits is obviously 2^n . Although the states of the nodes in a Hopfield network are binary, the storage capacity P_{max} is considerably lower than that, and it is also lower than N, the number of nodes. In fact it is on the order of 0.15N. Why is that the case? Let us define a quantity

$$C_i^\nu = -\xi_i^\nu \frac{1}{N} \sum_{j=1}^{N} \sum_{\mu=1,\mu\neq\nu}^{p} \xi_i^\mu \xi_j^\mu \xi_j^\nu \tag{57}$$

If $C_i^\nu < 0$, the crosstalk term has the same sign as the desired ξ_i^ν. Thus, in formula (56) h_i^ν always has the same sign as ξ_i^ν and therefore ξ_i^ν remains (is stable). We can define:

$$P_{error} = P(C_i^\nu > 1) \tag{58}$$

the probability that some bit in the pattern will not be stable (because in this case the crosstalk term changes the sign and the bit is flipped). It can be shown that the C_i^ν have a binomial distribution (since they are the sum of random numbers (-1,+1)).

Thus

$$P_{error} = \frac{1}{\sqrt{2\pi\sigma}} \int_{+1}^{\infty} \exp(-\frac{x^2}{2\sigma^2})dx = \frac{1}{2}[1 - erf(\frac{1}{\sqrt{2\sigma^2}})] = \frac{1}{2}[1 - erf(\frac{N}{2p})^{\frac{1}{2}}] \tag{59}$$

We can use table in figure 7. For example, if the error must not exceed 0.01 then the maximum number of patterns that can be stored is 0.185N. In fact, this is only the initial stability - matters may change over time, i.e. there can be avalanches. The precise number of patterns that can be stored in a Hopfield network depends on a variety of factors, in particular the statistical characteristics of the to-be-stored patterns. We will not go further into this topic; it is not so important. What does matter is the fact that there is a tradeoff between the storage capacity and the

P_{error}	$\frac{p_{\max}}{N}$
0.001	0.105
0.0036	0.138
0.01	0.185
0.05	0.37
0.1	0.61

FIGURE 7. Determining the maximum number of patterns that can be stored in a Hopfield network, given that the error must not exceed P_{error}.

desiderable properties of an associative memory. The interested reader is referred to [**Hertz et al., 1991**] or [**Amit, 1989**].

Note that the crosstalk term is zero if the patterns ξ_i^ν are orthogonal, i.e. $\sum_j \xi_j^\nu \xi_j^\mu = 0, \mu \neq \nu$
in which case $w_{ij} = 0$. In other words, this is no longer an associative memory. Thus, the capacity must be lower than N for the memory to be useful. We looked at some estimates above.

1.6. The energy function. One of the fundamental contributions of Hopfield was the introduction of an *energy function* into the theory of neural networks. Metaphorically speaking, this idea is the enabler for making a connection between the physical world (energy) and the one of information processing or signal processing in neural networks. The energy function H is defined as:

$$H = -\frac{1}{2}\sum_{i,j} w_{ij} S_i S_j \tag{60}$$

The essential property of the energy function is that it always decreases (or remains constant) as the system evolves according to the dynamical rule (49). Thus, the attractors (memorized patterns) in figure 5.1 correspond to local minima of the energy surface. This is a very general concept: in many systems there is a function that always decreases during the dynamical evolution or is minimized during an optimization procedure. The best known term for such functions comes from the theory of dynamical systems, the *Lyapunov function*. In physics it is called *energy function*, in optimization *cost function or objective function*, and in evolutionary biology, *fitness function*.

It can be shown that with the Hopfield dynamics the energy function always decreases. *This implies that memory content, stored patterns, represent physical minima of the energy function.* We have thus found a connection between physical quantities and information, so to speak.

Hopfield (1982) suggested to assume that $w_{ij} = w_{ji}$, the main reason being that this facilitates mathematical analysis. If we use rule (55) the weights will automatically be symmetric. However, from a biological perspective, this assumption is unreasonable. For symmetric weights we can re-write (60) as

$$H = C - \sum_{(ij)} w_{ij} S_i S_j \tag{61}$$

where (ij) means all distinct pairs ij (35 is the same as 53). The ii terms have been added into the constant C. It can now be shown that the energy decreases under the Hopfield dynamics (update rule (49)). Let S_i^{t+1} be the new value of S_i for some particular unit i :

$$S_i^{t+1} = sgn(\sum_j w_{ij} S_j^t) \tag{62}$$

The energy is unchanged if $S_i^{t+1} = S_i^t$. In the other case $S_i^{t+1} = -S_i^t$. We can pick out the terms that involve S_i.

$$H^{t+1} - H^t = -\sum_{j \neq i} w_{ij} S_i^{t+1} (S_j^{t+1})' + \sum_{j \neq i} w_{ij} S_i^t S_j^t \tag{63}$$

$S_i^{t+1} = S_i^t$ except for $i = j$. Thus
$H^{t+1} - H^t = 2S_i^t \sum_{j \neq i} w_{ij} S_j^t = 2S_i^t \sum_j w_{ij} S_j^t - 2w_{ii}$

The first term is negative because of (62) and $S_i^{t+1} = -S_i^t$, and the second term is negative because rule (49) gives $w_{ii} = p/N \forall i$. Thus, the energy decreases at every time step and we can write:

$$\Delta H \leq 0.$$

Although for this consideration the $w_{ii} \geq 0$ can be omitted, they influence the dynamics of the "spurious states". Let us write

$$S_i^{t+1} = sgn(w_{ii} S_i^t + \sum_{j \neq i} w_{ij} S_j^t) \tag{64}$$

Assume that $w_{ii} \geq 0$ then if $w_{ii} > \sum_{j \neq i} w_{ij} S_j^t$:
$S_i^t = +1 \rightarrow S_i^{t+1} = S_i^t$ since the sign remains unchanged;
$S_i^t = -1 \rightarrow S_i^{t+1} = S_i^t$ since the magnitude of the negative term is larger. In other words, *both* are stable. This leads to additional "spurious states" which in turn leads to a reduction of the basins of attraction of the "desired" attractors.

1.7. The landscape metaphor for the energy function. In this section we largely follow [**Amit, 1989**]. Remember that in chapter 4 we had defined an error function and visualized the dynamics in weight space as a point moving on the error surface. Now we define an energy function and look at the dynamics in state space. In the case of the error function we were interested in finding the global minimum, which corresponds to the weight matrix with the best overall performance. While we are still interested in finding the global minimum, the focus is on reliably storing as many patterns as possible, which then correspond to local minima of the energy function.

The most elementary landscape metaphor for retrieval from memory, for classification, for error correction, etc., can be represented as a one-dimensional surface with hills and valleys, as shown in figure 8. The stored memories are the coordinates

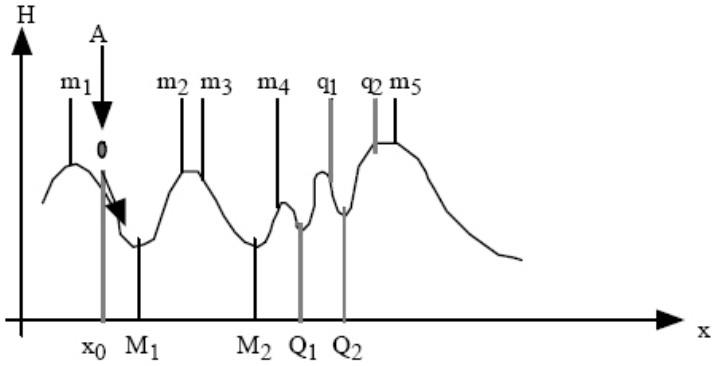


FIGURE 8. The one-dimensional landscape metaphor for associative, content-addressable memory. M_1 and M_2 are memories, Q_1 and Q_2 spurious states, $m_1, \ldots, m_5$ are maxima delimiting basins of attraction (after [**Amit, 1989**], p. 82).

of the bottoms of the valleys, M_1, M_2. Stimuli are "drops" which fall vertically on this surface, each stimulus carries with it x_0 - the coordinate along the horizontal axis of the point it starts from. The dynamical route of the "drop" can be imagined as frictional gliding - a point moving on a sticky surface. After a "drop" arrives at the bottom of a valley, it will stay there. The minima are therefore fixed point attractors. The coordinate of the particular minimum arrived at is called *the memory retrieved by the stimulus*. All stimuli between m_1 and m_2, for example, will retrieve the same memory, M_1. The fact that they all retrieve the same memory is referred to as *associative recall*, or *content addressability*. The range is the *basin of attraction* of the particular memory. The points lying within a particular basin of attraction can also be viewed as constituting an entire class of stimuli.

1.8. Perception errors due to spurious states - possible role of noise. As mentioned earlier, whenever memories are stored in a network as attractors, they are always accompanied by the appearance of spurious attractors, such as Q_1 and Q_2 . This can lead to errors of recognition or recall. However, the situation is not so bad because the spurious attractors are typically at higher levels in the landscape. Often the barriers are lower than for actual memory. Thus, with the addition of a certain noise level, the spurious states can be destabilized while the stored memories remain good attractors. Once again, we see that noise can indeed be very beneficial, not something to get rid of.

1.9. Simulated annealing. Especially when using Hopfield networks for optimization purposes (see below), we may be interested in finding the lowest energy state in the system. This can be done by *simulated annealing*. The states of the nodes can be made dependent on "temperature" by introducing stochastic nodes. Stochastic nodes have a certain probability of changing state spontaneously, i.e. without interference from the outside.

$$P(S_i \to -S_i) = \frac{1}{1+\exp(\beta \Delta H_i)}, \beta \simeq \frac{1}{T} \tag{65}$$

The higher the temperature, the higher the probability of state change. ΔH_i is the energy change produced by such a change. The smaller this change, the higher the probability of state change. The idea is to start with a high temperature T and to gradually lower it. This helps escaping from local minima, e.g. spurious attractors. With this procedure low energy states can be found, but the procedure remains stochastic, so there is no guarantee that at the end the system will be found at a strict global minimum. Moreover, simulated annealing is computationally extremely expensive which is why it is of little practical relevance.

1.10. Solving the traveling salesman problem. The traveling salesman problem, as an example of a hard optimization problem, has been successfully solved using Hopfield nets. It turns out that using stochastic units leads to better solutions, but the procedure is very time-consuming, i.e. there are high computational costs involved.

Very briefly, we have N points, the cities. The task is to find the minimum-length closed tour that visits each city once and returns to its starting point. Originally, the Hopfield solution to the traveling salesman problem has been proposed by [**Hopfield and Tank, 1985**,**Hopfield and Tank, 1986**]. Most NN textbooks discuss this problem.

Applying Hopfield networks to solve the traveling salesman problem is relatively tricky and it seems that the solution proposed by Hopfield and Tank ([**Hopfield and Tank, 1985**]) does not work very well, for example, it doesnt scale well to problems sizes of real interest. Wilson and Pawley ([**Wilson and Pawley, 1988**]) examined the Hopfield and Tank algorithm and they tried to improve on it, but failed. In the article, they also provide the reason for this failure.

In summary, Hopfield nets, because of their characteristics as associative or content-addressable memories with much psychological appeal, have had a great theoretical impact, they have been applied to problems in physics (e.g. spin-glass models), and variations of them have been used for brain modeling. However, their applicability in practice has been relatively limited.

1.11. Boltzman machines. Boltzman machines can be seen as extension of Hopfield nets. In contrast to Hopfield nets, a distinction is made between hidden and visible units, similarly to MLPs. The hidden units can be divided into input and output units. The goal is to find the right connections to the hidden units without knowing from the training patterns what the hidden units should represent. Because Boltzman machines are computationally also very expensive, they are only of little practical use. We do not discuss them here.

1.12. Extensions of the Hopfield Model: Continuous-valued units. Hopfield models have been extended in various ways, most prominently by introducing continuous-valued units with dynamics described by differential equations, but also by introducing mechanisms by which the system can transition to other attractor states. We only provide the general idea of how such systems can be described. We only consider units that have the following property (the standard node characteristic):

$$V_i = g(u_i) = g\left(\sum_j w_{ij} V_j\right) \tag{66}$$

The dynamics of the activation level of the units can be described by the following equation:

$$\tau_i \frac{dV_i}{dt} = -V_i + g(u_i) = -V_i + g\left(\sum_j w_{ij} V_j\right) \tag{67}$$

where τ_i are suitable time constants. And input term ξ_i can be added to the right-hand side of the equation.

If $g(u)$ is of the sigmoid type and the w_{ij}s are symmetric, the solution $V_i(t)$ always settles down to a stable equilibrium solution.

Just as in the discrete case, and energy function can be defined, and it can be shown that it always decreases under the dynamics described by equation (67). It is the continuous version of Hopfield nets that has been used by Hopfield and Tank in their solution to the traveling salesman problem.

2. Other recurrent network models

In the previous section we discussed an important type of recurrent network, the Hopfield net and some of its variations. The essential point is that recurrent networks always embody a dynamics, a time course. Other examples of recurrent networks are the "brain-state-in-a-box" model (for a very detailed description, see [**Anderson, 1995**], chapter 15), the "bidirectional associative memories" ([**Kosko, 1992**], pp. 63-92). In this section we discuss recurrent backpropagation, CTRNNs, and finally, echo-state networks.

2.1. Recurrent backpropagation. So far we have been discussing the storage and retrieval of individual patterns. If we are interested in behavior in the real world, we normally do not work with one pattern but with a sequence of patterns. Hopfield nets have been extended to include the possibility for storing and retrieving sequences (e.g. [**Kleinfeld, 1986**]), and for generating periodic motion ([**Beer and Gallagher, 1992**]). Here, we only have a brief look at recurrent backpropagation. A popular way to recognize (and sometimes reproduce) sequences has been to use partially recurrent networks and to employ context units.

Figure 9 shows a number of different architectures. 9 (a) is the suggestion by [**Elman, 1990**]. The input units are separated into input units and context units. The context units hold a copy of the activation of the hidden units from the previous time step. The modifiable connections are all feedforward and can be trained by conventional backpropagation methods. Figure 9 (b) shows Jordan's approach [**Jordan, 1986**]. It differs from 9 (a) in that the output nodes are fed back into the context nodes and the context nodes are connected to themselves, thus providing a kind of short-term memory. The update rule is:
$C_i(t+1) = \alpha C_i(t) + O_i(t)$
where O_i are the output units and α is the strength of the self-connections. Such self-connecting units are sometimes call *decay units, leaky integrators*, or *capacitive units*. With *fixed* input, the network can be trained to generate a set of output

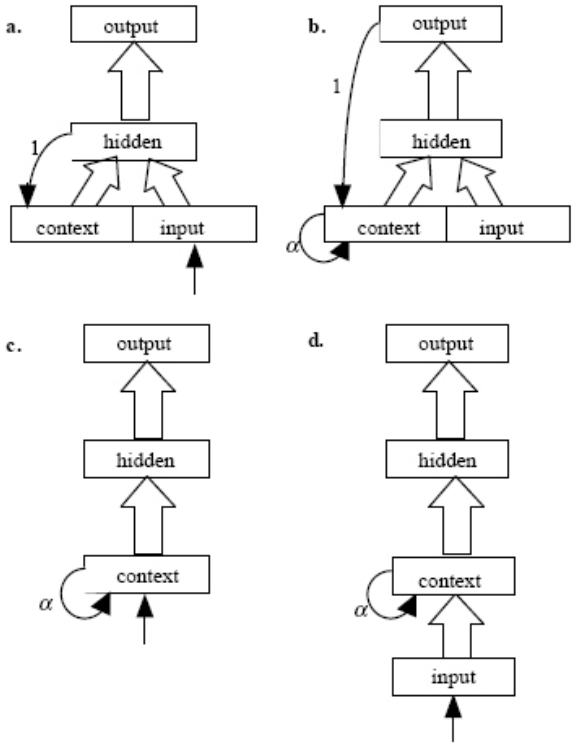


FIGURE 9. Architectures with context units. Single arrows represent connections only from the i-th unit in the source layer to the i-th unit in the destination layer, whereas the wide arrows represent fully connected layers (after [**Hertz et al., 1991**]).

sequences with different input patterns triggering different output sequences. With an *input sequence*, the network can be trained to recognize and distinguish different input sequences.

Figure 9 (c) shows a simpler architecture that can also perform sequence recognition tasks. In this case, the only feedback is from context units to themselves. This network can recognize and distinguish different sequences, but is less well suited than 9 (a) and (b) to generating or reproducing sequences. Figure 9 (d) is like (c) but the weights from the input to the context layer are modifiable, and the selfconnections are not fixed but trained like all the other connections. Like 9 (c) this architecture is better suited to recognizing sequences than to generating or reproducing them.

2.2. Continuous time recurrent neural networks (CTRNN). The networks that we have considered so far have mostly been based on discrete time scales and their activation has been synchronized to operate in a discrete manner: $a_i(t+1) = a_i(t) + \sum w_{ij} a_j(t)$. With this kind of abstraction very interesting results have been achieved as described so far in this script. However, it is clear that biological brains are not clocked in this way, but change continuously, rather than at

discrete points in time. CTRNNs have become popular in the field of evolutionary robotics where the weights are normally determined using artificial evolution. The task of finding the proper weights is more complicated than in standard feedforward networks, because there is a complex intrinsic dynamics (due to the recurrent connections). This intrinsic dynamics can be exploited by the robot, as this intrinsic activation can be used to implement a kind of short term or working memory.

CTRNNs can be formally described as:

$$\tau y_i' = -y_i + \sum_{j=1}^{N} w_{ij}\sigma(y_j - \theta_j) + \sum_{k=1}^{S} s_{ik}I_k, i = 1, 2, \ldots, N$$

where y is the state of the neuron, τ is the time constant of the neuron, N is the total number of neurons, w_{ij} gives the strength of the connection, σ is the standard sigmoid activation function, θ is a bias term, S is the number of sensory inputs, I_k is the output of the k^{th} sensor, and s_{ik} is the strength of the connection from sensor to neuron.

Neurons that have the characteristics described here, are also called "leaky integrators", i.e. they partially decay and partially recycle their activation and can – in this way – be used for short-term memory functions.

In a fascinating set of experiments, the computational neuroscientist Randy Beer evolved simulated creatures that had the task to distinguish between a diamond and a circle using information from "ray sensors", i.e. sensors that measure presence or absence in a particular direction (Beer‘s agent had 7 such ray sensors). More precisely, Beer used a genetic algorithm to determine the weights of the recurrent part of the neural network, because it is way hard to find proper learning rules for recurrent networks. The agent was equipped with a CTRNN type neural network: the input layer was attached to the ray sensors, the output layer consisted of nodes telling the agent to move left or right, and the hidden layer consisted of a fully recurrent network. From the "top" an object, either a diamond or a circle was dropped into the scene. If the object was a diamond, the agent had to move away from it, if it was a circle it had to move towards it. Beer used a genetic algorithm to determine the weights. The best agents, i.e. the ones that could perform the distinction most reliably were the ones that moved left and right a few times, before moving either towards or away from the object. In other words, they engaged in a sensory-motor coordination. The purpose of this sensory-motor coordination is to generate the additional sensory stimulation needed. This is compatible with the principle of sensory- motor coordination. For more detail, see [**Beer, 1996**] or [**Pfeifer and Scheier, 1999**].

For additional applications of CTRNNs see, for example, [**Ito and Tani, 2004**, **Beer, 1996**].

2.3. Echo state networks (ESN). Nonlinear dynamical systems are very popular both in science and engineering because the physical systems they are intended to describe are inherently nonlinear in nature. The analytic solution to nonlinear systems is normally hard to find and so the standard approach lies in the qualitative and numeric analysis. The learning mechanism in the biological brain is intrinsically a nonlinear system. To this aim, the echo state network models the nonlinear behavior of the neurons thus incorporating an artificial recurrent neural network(RNN). Its peculiarity with respect to other kinds of artificial RNN resides

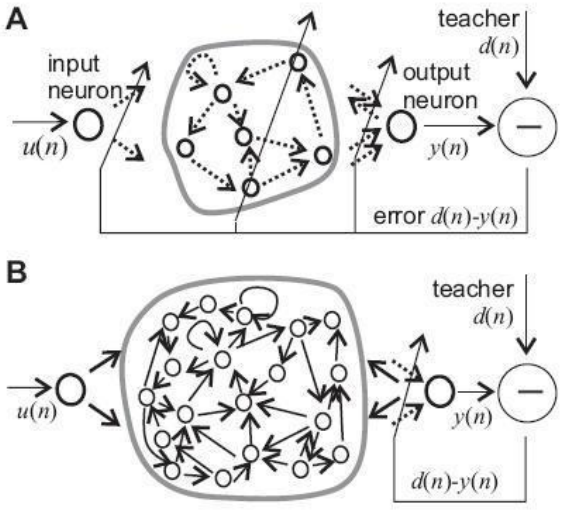


FIGURE 10. Two approaches to RNN learning: (A) Schema of previous approaches to RNN learning. (B) Schema of ESN approach. Solid bold arrows: fixed synaptic connections, dotted arrows: adjustable connections. Both approaches aim at minimizing the error d(n) y(n), where y(n) is the network output and d(n) is the "teacher" time series observed from the target system.

in the high number of neurons within the RNN(order of 50 to 1000 neurons) and in the locality of the synapses which are being adjusted by learning(only those that link the RNN with the output layer). Due to this structure, the ESN benefits both from the high performance and dynamics exhibited by a RNN and from the linear training complexity, respectively. An ESN is depicted in Figure 10 referred from [**Jaeger and Haas, 2004**].

The underlying mathematics of ESNs consists of:

(1) the state equation:
$x(n+1) = \tanh(Wx(n) + w_{in}u(n+1) + w_{fb}y(n) + v(n)$,
where $x(n+1)$ is the network state at discrete time $n+1$, W is the N^2-size matrix of the RNN's internal weights, w_{in} is the N-size vector of input weights, $u(n+1)$ is the input vector at the current time, w_{fb} is the weight vector from the output to the RNN, $y(n)$ is the output vector obtained previously and $v(n)$ is the noise vector;

(2) the output equation:
$y(n) = \tanh w_{out}(x(n), u(n))$,
where w_{out} is a (N+1)-sized weight vector to the output layer.

The neurons in the artificial recurrent neural network of the ESN are sparsely connected, reaching a value of 1% interconnectivity. This decomposes the RNN into loosely coupled subsystems and ensures a rich variation within it. Due to the recurrency and the feedback received from the output neurons, a bidirectional dynamical interplay unfolds between the internal and the external signals. The excitation in

the internal layer is viewed as an echo to the signals coming from the output layer.

The ESN represents a powerful tool for time series prediction, inverse modeling (e.g. inverse kinematics in robotics), pattern generation, classification (on time series) and nonlinear control. The biological features that the ESN designs make it suitable as well as a model for prefrontal cortex function in sensory-motor tasks, for models of birdsong and of cerebellum, etc.

In this chapter we have looked at recurrent networks. Because of the loops, recurrent networks have an intrinsic dynamics, i.e. their activation changes even if there is no input, a characteristic that is fundamentally different from the previously discussed feed-forward networks. These properties lead to highly interesting behavior and we need to apply the concepts and terminology of complex dynamical systems to describe their behavior. We have only scratched the surface, but this kind of network is highly promising and reflects at least to some extent properties of biological networks. Next we will discuss non-supervised learning.

CHAPTER 6

Non-supervised networks

In the previous chapters we studied networks for which the designer determined beforehand the kinds of patterns, mapping, and functions they had to learn. We called it the large class of supervised networks. While feedforward networks do not have an interesting dynamics in state space - the patterns are merely propagated from input layer to output layer - network with recurrent connections do. An important example of the latter kind are the Hopfield nets that can be used as associative memories. In all of the networks discussed so far, the physical arrangement did not matter: notions like "closer" or "further away" did not make sense. In other words, there was no metric defined on them. The only notion of "*distance*" between nodes has been the fact, that for example in feedforward networks, input nodes are "closer" to the nodes in the hidden layer, than to the ones in the output layer. In Hopfield and winner-take-all networks, the step length is always one. In some of the networks that we will introduce here - the topographic feature maps - distance neighborhood relationships play an important role.

The fascinating point about the networks in this chapter is that learning is not supervised (or unsupervised), i.e. the designer does not explicitly specify mappings to be learned. The training sets are no longer pairs of input-desired output, but simply patterns. Typically, these networks then cluster the patterns in certain ways. The clusters in turn, can be interpreted by the designer as categories.

We start with a simple form of non-supervised networks, competitive schemes and look into geometric interpretations. This is followed by a discussion of ART, Adaptive Resonance Theory, a neurally inspired architecture that is on the one hand unsupervised and on the other contains recurrent network components. Then we introduce topographic maps, also called Kohonen maps that constitute a kind of competitive learning. The chapter ends with a note on Hebbian learning, which is a form of non-competitive nonsupervised learning.

1. Competitive learning

The learning mechanisms that we will discuss in this section are called "competitive" because, metaphorically speaking, the nodes "compete" for input patterns – one of them will be the winner.

1.1. Winner-take-all networks. Consider the following network (figure 1). The output nodes in this network "compete" for the activation: once they have an advantage, they inhibit the other nodes in the layer through the negative connections and activate themselves through positive connections. After a certain time only one node in the output layer will be active and all the others inactive. Such a network layer is called a *winner-take-all network*. The purpose of such a network is to categorize an input vector $\underline{\xi}$. The category is given by a node in the output

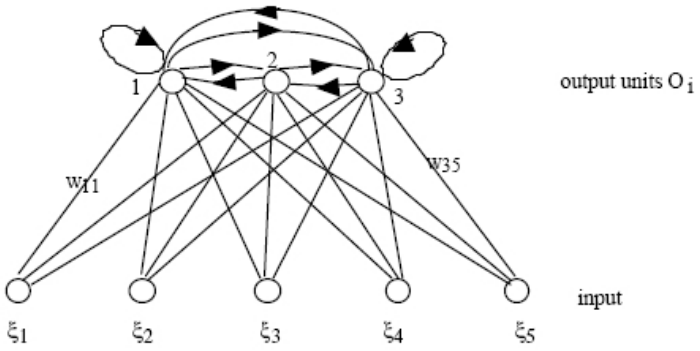


FIGURE 1. A simple competitive learning network. The nodes in the output layer are all connected to one another by negative - inhibitory - connections. The self-connections are positive - excitatory. A network layer with this connectivity is called a *winner-take-all network*. All output nodes are connected to themselves (not all shown in the figure).

layer. Such a node is sometimes called a grandmother cell. Here is why: Assume that the input vector, the stimulus, originates from a sensor (e.g. a CCD camera) taking a picture of your grandmother. If this image - and others that are similar to this one - are mapped onto a particular node in the output layer, the node can be said to represent your grandmother, it is the *grandmother cell*. The point is that the network has to cluster or categorize inputs such that similar inputs lead to the same categorization (generalization).

In the simplest case, the input nodes are connected to the output layer with excitatory connections $w_{ij} \geq 0$. The winner is normally the unit with the largest input.

$$h_i = \sum_j w_{ij}\xi_j = \underline{w_i} \cdot \underline{\xi} \tag{68}$$

If i^* is the winning unit, we have

$$\underline{w_{i^*}} \cdot \underline{\xi} \geq \underline{w_i} \cdot \xi \forall i \tag{69}$$

If the weights for each output unit are normalized, $|w_i| = 1$, i.e. if the sum of the weights leading to output unit O_i sum to 1, for all i , this is equivalent to

$$|\underline{w_{i^*}} - \underline{\xi}| \leq |w_i - \underline{\xi}| \forall i \tag{70}$$

This can be interpreted as follows: the winner is the unit for which the normalized weight vector $\underline{w}$ is most similar to the input vector $\underline{\xi}$. How the winner is determined is actually irrelevant. It can either be found by following the network dynamics, simply using the propagation rule for the activation until after a number of steps only one output unit will be active. However, this approach is computationally costly and the shortcut solution of simply calculating h_i for all output units

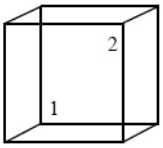


FIGURE 2. The Necker cube. Sometimes corner 1 is seen as closer to the observer, in which case the cube is seen from below, sometimes corner 2 is seen as closer to the observer, in which case the cube is seen from above.

and finding the maximum is preferable. If the network is winner-take-all, the node with the largest h_i is certain to win.

Psychologically speaking, winner-take all networks - or variations thereof - have a lot of intuitive appeal. For example, certain perceptual situations like the Necker cube shown in figure 2 can be nicely modeled. There are two ways in which the cube can be seen. One interpretation is with corner 1 in the front, the other one with corner 2 in the front. It is not possible to maintain both interpretations simultaneously. This can be represented by a winner-take all network that classifies the stimulus shown in figure 2: If the nodes in the output layer correspond to interpretations of the cube, it is not possible that two are active simultaneously. There is the additional phenomenon that the interpretation switches after a while. In order to model this switching phenomenon, extra circuitry is needed (not discussed here).

1.2. The standard competitive learning rule. The problem that we have to solve now is how we can find the appropriate weight vectors such that clusters can be found in the input layer. This can be achieved by applying the competitive learning rule:

- Start with small random values for the weights. It is important to start with random weights because there must be differences in the output layer for the method to work. The fancier way of stating this is to say that the symmetry must be broken.
- A set of input patterns $\underline{\xi^\mu}$ is applied to the input layer in random order.
- For each input $\underline{\xi^\mu}$ find the winner i^* in the output layer (by whatever method).
- Update the weights $\underline{w_{i^*j}}$ for the winning unit only using the following rule:
  $\Delta w_{i^*j} = \eta(\frac{\xi_j^\mu}{\sum_{j=1}^\mu \xi_j^\mu} - w_{i^*j})$
  This rule has the effect that the weight vector is moved closer to the input vector. This in turn has the effect that next time around, if a similar stimulus is presented, node i^* has an increased chance of being activated.

If the inputs are pre-normalized, the following rule, the so-called *standard competitive learning rule* can be used:

$$\Delta w_{i^*j} = \eta(\xi_j^\mu - w_{i^*j}) \tag{71}$$

In equation (71) it can be directly seen that the weight vector is moved in the direction of the input vector. If they coincide, there is no learning taking place.

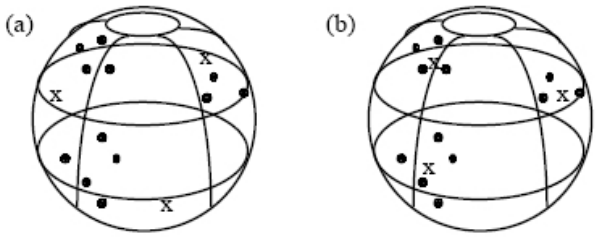


FIGURE 3. Geometric interpretation of competitive learning. The dots represent the input vectors, the x-s the weights for each of the three units of figure 1. (a) is the situation before learning, (b) after learning: the x-s have been "pulled into the clusters".

Note that the learning rules only apply to the winner node; the weights of the other nodes are not changed.

If we have binary output units, then for the winner node i^* we have $O_{i^*} = 1$, and for all the others $O_i = 0, i \neq i^*$. Thus, we can re-write (71) as

$$\Delta w_{ij} = \eta O_i(\xi_j^\mu - w_{ij}) \tag{72}$$

which looks very much like a Hebbian rule with a decay term (or a forgetting term, given by w_{ij}) (Hebbian learning: see later in this chapter).

1.3. Geometric interpretation. There is a geometric interpretation of competitive learning, as shown in figure 3.

The weight vectors $\underline{w_i} = (w_{i1}, w_{i2}, w_{i3})$ can be visualized on a sphere: the point on the sphere is where the vector penetrates the sphere. If $|\underline{w}| = 1$ the end points of the weight vectors are actually on the sphere. The same can be done for the input vectors. Figure 3 illustrates how the weight vectors are pulled into the clusters of input vectors. Note the number of xs is the maximum number of categories the network can represent.

1.4. Vector quantization. (In this section we largely follow [**Hertz et al., 1991**]).

One of the most important applications of competitive learning (and its variations) is *vector quantization*. The idea of vector quantization is to categorize a given set or a distribution of input vectors $\underline{\xi_i}$ into M classes, and to represent any vector by the class into which it falls. This can be used, for example, both for storage and for transmission of speech and image data, but can be applied to arbitrary sets of patterns. The vector components ξ_j^μ are usually continuous-valued. If we transmit or store only the index of the class, rather than the input vector itself, a lot of storage space and transmission capacity can be saved. This implies that first a codebook has to be agreed on. Normally, the classes are represented by M prototype vectors, and we find the class of a given input by finding the nearest prototype vector using the ordinary Euclidean metric. This procedure divides the input space into so-called Voronoi Tessalations as illustrated in figure 4.

The term vector quantization means that the continuous vectors ξ_j^μ are represented by the prototype vector only, in other words by the class index (which

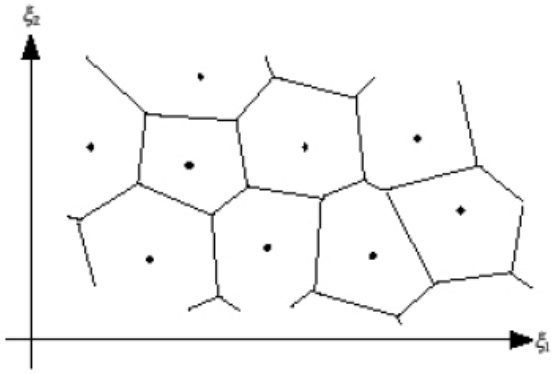


FIGURE 4. Voronoi tessalations. The space is divided into polyhedral regions according to which of the prototype vectors (the black dots) is closest. The boundaries are perpendicular to the lines joining pairs of neighboring prototype vectors.

is discrete or quantized). The translation into neural networks is straightforward. For each input vector ξ_j^μ find the winner, the winner representing the class. The weights $\underline{w_i}$ are the prototype vectors. We can find the winner i^* by

$$|w_{i^*} - \underline{\xi}| \leq |w_i - \underline{\xi}| \forall i \tag{73}$$

which is equivalent to maximizing the inner product (or scalar product) $\underline{w_i} \cdot \underline{\xi}$ if the weights are normalized. In other words, we can use the standard competitive learning algorithm.

A problem with competitive learning algorithms are the so-called "dead units", i.e. those units with weight vectors that are too far away from any of the input vectors that they never become active, i.e. they never become the winner and thus never have a chance to learn. What can be done about this?
Here are a few alternatives:

(1) Adding noise with a wide distribution to the input vectors.
(2) Initialize the weights with numbers drawn from the set of input patterns.
(3) Update of the weights in such a way that also the "losers" have a chance to learn, but with a lower learning rate η':
$$\Delta w_{ij} = \begin{cases} \eta(\xi_j^\mu - w_{i^*j}), & i^* \text{ winner} \\ \eta'(\xi_j^\mu - w_{ij}), & i \text{ loser} \end{cases}$$
This is called "leaky learning".
(4) Bias subtraction: The idea is to increase the threshold for those units that win often and to lower the threshold for those that have not been successful. This is sometimes metaphorically called a "conscience mechanism" in the sense that the winner nodes should, over time, develop a bad conscience because they always win and should let others have a chance as well. This could also be interpreted as changing the sensitivity of the various neurons. This mechanism is due to [**Bienenstock et al., 1982**].

(5) Not only the weights of the winner should be updated but also the weights belonging to a neighbor. This is in fact, the essence of Kohonen feature mapping (see below).

It is generally also a good idea to reduce the learning rate successively. If η is large there is broad exploration, if η is small, the winner does not change any longer, but only the weights are tuned.
Examples are: $\eta(t) = \mu_0 t^{-\alpha}(\alpha \leq 1)$, or $\eta(t) = \eta_0(1 - \alpha t), \alpha =$ e.g. 0.1
An energy function can also be defined (see, e.g. [**Hertz et al., 1991**], p. 222), but will not be discussed here.

2. Adaptive Resonance Theory

Competitive learning schemes as presented above suffer from a basic flaw: The categories they have learned to recognize may change upon presentation of new input vectors. It has been shown that, under specific conditions, even the repeated presentation of the same set of input vectors does not lead to stable categories but to an oscillatory behavior (i.e. the vectors representing two different categories are slowly shifted and eventually even completely interchanged). This is an instance of a fundamental problem, namely *Grossberg's stability-plasticity dilemma* (which in turn is an instance of the more general diversity/compliance issue [**Pfeifer and Bongard, 2007**]): How can a system of limited capacity permanently adapt to new input without washing away earlier experiences? [**Carpenter and Grossberg, 2003**] illustrate this by an example: Assume that you are born in New York and move to Los Angeles after your childhood. Even if you stay there for a long time and despite the fact that you will learn quite many new streets and secret paths, you will never forget where you have lived in New York and you always will find the way to your earlier home. You preserve this knowledge during your whole life. Translated to neural networks, this indicates that we search for a neural network and an adaptation algorithm that improves continuously when being trained in a stationary environment but, when confronted with unexpected input, starts categorizing the new data from scratch without forgetting already learnt content. Such a scheme was introduced and later refined by Carpenter and Grossberg in a series of papers [**Grossberg, 1987**, **Carpenter et al., 1987**, **Carpenter et al., 1991a**, **Carpenter and Grossberg, 1990**, **Carpenter et al., 1991b**] (for an overview, see e.g.. [**Carpenter and Grossberg, 2003**].

Pondering how to overcome Grossberg's dilemma leads to the insight that a solution of the problem requires answering a closely related, technical question: How many output nodes, representing possible categories, should we use? Working with a fixed number of nodes right from the start may lead to an exhaustion of these nodes in a first stationary environment by a fine-grained categorization, which may well be of no value at all (Remember that we are aiming for an *unsupervised* network), and prohibits the formation of novel categories in a changed situation, or only allows it at the cost of overwriting existing ones. It would be more sensible to use a procedure with a dynamically changing number of categories; a new category is only formed (an output node is recruited and "committed") if an input event cannot be assigned to one of the already defined categories. More technically, the assignment to an existing node or, if not possible, the creation of a new one relies on some similarity criterion. Given such a measure for similarity, of a network that

resolves the stability-plasticity dilemma in a satisfactory way might work as follows (as in the case of ART):

(1) The input is sufficiently similar to an already existing category: In order to improve the network, the representation of the existing category is adapted in a manner that the recognition of the given input is amplified (the representation is refined), but in such a way that the original category is preserved. The algorithm to be presented below achieves this goal by changing the weights but at the same time avoiding washing out previously formed categories.
(2) There is no sufficiently similar category: The input serves as an instance for a new category, which is added by committing an uncommitted output node.
(3) No assignment is possible, but no more categories (output nodes) are available: In a practical implementation, this case has to be included but it is not relevant in a theoretical investigation, where we can always assume the number of nodes to be finite but taken from an inexhaustible reservoir.

The similarity criterion employed as well as the way in which the representations of the categories are adapted is of course implementation dependent. The specific way in which the similarity criterion is implemented in the work of [**Grossberg, 1987**] justifies the term "Resonance", which can be taken quite literally. In the algorithm presented below, however, resonance is rather a metaphor.

2.1. ART1. ART1 is based on an algorithm mastering the stability-plasticity dilemma for binary inputs ξ in a network with a structure as presented in Fig. 1 in an acceptable way. The index i designates the outputs, each of which can be either enabled or disabled and is represented by a vector w_i (the components of these w_i, namely w_{ij}, are used as weights for the connections, as in previous sections).

One starts with $w_i = 1$, i.e. vectors with components all set to one. These w_i are uncommitted (not to be confused with disabled outputs) and their number should be large. The algorithm underlying ART1 then works as follows (we follow here the presentation of [**Hertz et al., 1991**]):

(1) Enable all output nodes.
(2) Formation of a hypothesis: Find the winner i^* among all the enabled output nodes (exit if there are none left). The winner is defined as the node for which $T_i = \underline{w}_i \cdot \xi$ is largest. Thereby, $\underline{w}_i$ is a normalized version of w_i, given by

$$\underline{w}_i = \frac{w_i}{\epsilon + \sum_j w_{ij}} \tag{74}$$

The small number ϵ is included to break ties, selecting the longer of the two w_i which both have all their bits in ξ. Note that an uncommitted node wins, if there is no better choice.
(3) Check of hypothesis: Test whether the match between ξ and w_i is good enough by computing the ratio

$$\tau = \frac{w_{i^*} \cdot \xi}{\sum_j \xi_j} \tag{75}$$

This is the fraction of bits in ξ that are also in w_{i^*}. If $\tau \geq \rho$, where ρ is the *vigilance parameter*, there is resonance; go to step 4. If, however, $\tau < \rho$, the prototype vector w_{i^*} is rejected. Disable i^* and go to step 2.

(4) Adjust the winning vector w_{i^*} by deleting any bits that are not also in ξ. This is a logical AND operation and is referred to as masking the input.

This algorithm continues to exhibit plasticity until all output units are used up. It can be shown that all weight changes cease after a finite number of presentations of a fixed number of inputs ξ. This is possible because logical AND is an irreversible operation (once a bit has been "washed out", it cannot be re-established but is gone forever).

The weights w_{ij} can be viewed as the representatives of the categories recognized by the ART1-system. From this point of view, the ART1-algorithm is a recurrent process of bottom-up formation of a hypothesis and subsequent top-down testing (the reason for employing the terms "bottom-up" and "top-down" will become obvious below, when we discuss a network implementation of ART1). Selecting an output node (by a winner-take-all scheme) by choosing i^* such that $T_i = \underline{w}_i \cdot \xi$ is maximal can be interpreted as forming the hypothesis that ξ is in the category represented by w_{i^*}. Assuming ϵ to be small and interpreting $w_{i^*j} = 1$ as "the category i^* exhibits property j", the hypothesis is just formed by choosing that category for which the fraction of properties of the representative of the category shared by the input ξ is maximal. Note that additional properties of the input not shared by the representative of the category don't influence the result. The input ξ is used in a "bottom-up" manner to calculate the T_i. After selecting the winner among these T_i's, the respective weights w_{ij} are used to determine $\tau = \frac{w_{i^*} \cdot \xi}{\sum_j \xi_j}$ which in turn has to satisfy the vigilance criterion. Note that τ measures the fraction of properties of ξ that are shared by the representative of the category i^*. In contrast to the formation of a hypothesis, in the "top-down" test, all properties of ξ count.

ART1 is *approximative match* and not *mismatch driven*, which means that memories are only changed when the actual input is close enough to internal expectations. If this is not the case, a new category is formed. Match driven learning is the underlying reason for the stability of ART and contrasts with error based learning that changes memories in order to reduce the difference between input and target output, rather than searching for a better match (remember the repeated search in ART1 where categories are disabled if they don't pass the criterion imposed by the vigilance parameter).

ART has a further interesting property: It is capable of *fast learning* which implies that one single occurrence of an unexpected input immediately leads to the formation of a new category. In this respect, ART resembles the learning abilities of the brain. Taking this into account, the importance of dynamic allocation of output nodes becomes clear. No allocation takes place in a stable environment, although the recognition of categories is refined upon the presentation of input, but an uncommitted node may be committed upon the single presentation of input from a new environment.

In what follows, we discuss a (sketch of an) implementation of the ART1 algorithm in an actual network. The goal of this is twofold: First, we clarify the different roles of the initial bottom-up formation of a hypothesis and the subsequent top-down comparison of this hypothesis with the given input. Second, we aim to give an impression of some of the necessary (still not all) regulation units

needed in a network implementation. In contrast to an algorithm realized by a computer program in which variables can be assumed to have only specific values (e.g. 0/1) and processing is sequential, a network implementation requires taking care of all system states (including transient, non-integer ones) that can emerge during processing. Furthermore, one has to consider that all system units are permanently interacting and not "called" in a sequential order.

The network, schematically given by Fig. 5, consists of two layers, the input and the feature layer. The outputs O_i of the latter represent the categories. Both layers are fully connected in both directions. We adopt the convention that a connection between the output V_j of the input layer and the output O_i of the feature layer is equipped with the weight w_{ij} independent of its direction (i.e. $w_{ij}^{\text{bottom-up}} = w_{ij}^{\text{top-down}}$). The input patterns are given by $\xi = (\xi_1, ..., \xi_N)$, A and R are threshold units and E emits a signal if ξ is changed. The values of O_i, V_j, A, R, ξ_j are either zero or one.

Assume for a moment the V_j to be given (we will explain later how they are calculated as a function of ξ). The feature layer consists of winner-takes-all units, which means that among the enablad O_i the one is chosen (set to 1, all other are 0) for which

$$T_i = \frac{\sum_j w_{ij}\xi_j}{\epsilon + \sum_j w_{ij}} \tag{76}$$

is maximal. This process is "bottom-up" in the sense that a result in the feature layer is based on the output of the input layer only.

The unit A is a threshold unit with threshold value 0.5 and threshold function $\sum_j \xi_j - N \sum_i O_i$: If no output is active, A is one, otherwise zero.

Having A and the O_i, we can calculate

$$h_j = \xi_j + \sum_i w_{ij} O_i + A. \tag{77}$$

This is used to determine V_j according to

$$V_j = \begin{cases} 1 & h_j > 1.5 \\ 0 & else \end{cases} \tag{78}$$

This is the so called "2/3 -rule" (which reads "two out of three" rule). In order to understand it, note that in equ. 77 at most one of the O_i can be non-zero. V_j is set to one if two out of the three inputs to equ. 77, namely A, ξ_j and one of the O_i's, are one. The role of A is to guarantee a sensible outcome of the threshold evaluation in equ. 78 even in those network states in which no hypothesis has been formed yet. Taking this into account, equ. 78 leads formally to

$$V_j = \begin{cases} \xi_j & A = 1 \\ \xi_j \wedge \sum_i w_{ij} O_i & A = 0 \end{cases} \tag{79}$$

Now, the reset signal R is discussed. If it is turned on while a winner is active, this winner is disabled and consequently removed from future competition until it is re-enabled. R is implemented as a threshold unit with

$$R = \begin{cases} 1 & \rho \sum_j \xi_j - \sum_j V_j > 0 \\ 0 & \text{else} \end{cases} \tag{80}$$

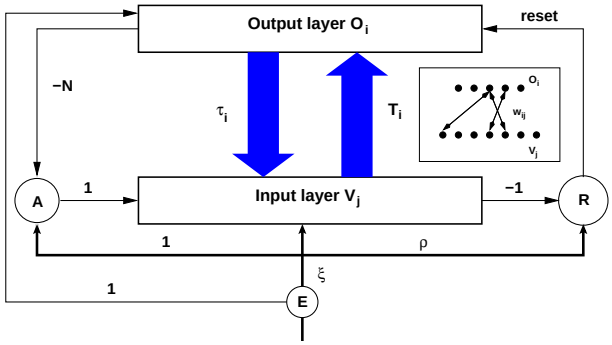


FIGURE 5. Sketch of a network implementation of the ART1 algorithm. Thick arrows represent buses, means multi-wires carrying multiple signals ($\xi = (\xi_1, ..., \xi_N)$) in parallel. The input layer gets the signal ξ and emits $V = (V_1, ..., V_N)$, which is constructed as described in the text. The signal V is then used calculate the inputs T_i of the feature layer, which in turn determines the according output $O = (O_1, ..., O_M)$ (formation of hypothesis). These outputs represent the different categories: They can take the values zero and one and are used for the calculation of τ which determines whether a hypothesis satisfies the vigilance criterion. Additionally, they are either enabled or disabled. The units A and R are threshold units necessary for the control of the network. Their function, as well as remaining signals and units, are explained in the text. The inset emphasizes the network nature of the structure and defines the weights w_{ij}.

This formula is easy to understand if one considers that the vigilance criterion reads $w_{i^*} \cdot \xi > \rho \sum_j \xi_j$ and takes into account equ. 78.

The weights are adapted according to

$$\frac{dw_{ij}}{dt} = \eta O_i (V_j - w_{ij}). \tag{81}$$

Finally, E is activated for a short time in case of a change of the presented input ξ. Its signal triggers the enbabling of all outputs in the feature layer and sets them the zero.

Analyzing this (simplified) network implementation shows that, in contrast to an abstract algorithm, several time scales play an important role (reaction times of different units, parameters such as η). The study of specific network implementations is, however, of interest because they yield indications to what extent ART-like structures can be regarded as models of actual neural structures. The reference to biological networks is one of the motivations for the choice of threshold units and relatively simple adaptation rules.

ART1 schemes can cope stably with an infinite stream of input and proved for example to be efficient in the recognition of printed text (mapped onto a binary grid), but there are also some drawbacks:

(1) ART1 showed to be susceptible to noise. This means that if input bits are randomly corrupted, the network may deteriorate quickly and irreversibly. This is the disadvantage of the adaptation of a representation by an irreversible operation such as AND.
(2) Destruction of an output node leads to the loss of the whole category it represents.

2.2. Further Developments. ART1 is restricted to binary input. This restriction is overcome in ART2, in which also continuous input is allowed [**Carpenter et al., 1987**, **Carpenter et al., 1991a**]. ART2 proved to be valuable in speech recognition. Further developments are given by ART3 [**Carpenter and Grossberg, 1990**] (the algorithm incorporates elements of the behavior of nerve cells), ARTMAP (a form of supervised learning by coupling two ART networks) and Fuzzy ART [**Carpenter et al., 1991b**]. FUZZY ART achieves the generalization to learning both analog and binary input patterns by replacing appearances of the AND operator in ART1 by the MIN operator of fuzzy set theory.

2.3. Fuzzy ART. (For a detailed presentation of fuzzy logic and neural networks, consult the book [**Kosko, 1992**]). Fuzzy ART provides a further possibility to extend the ART algorithm from binary to continuous inputs by employing operators from fuzzy instead of classical logic. Fuzzy logic is a well developed field of its own; here, we only present the definition of the operators required in the context of fuzzy ART:

(1) Assume X to be a set. A fuzzy subset A of X is given by a characteristic function $m_A(x) \in [0,1], x \in X$. One may understand $m_A(x)$ as a measure for the A-ness of x. Classical set theory is obtained by requiring $m_A(x) \in \{0,1\}$.
(2) The intersection of classical set theory $C = A \cap B$ is replaced by $C = A \bigwedge B$ with $m_C(x) = min(m_A(x), m_B(x))$.
(3) The union of classical set theory $C = A \cup B$ is replaced by $C = A \bigvee B$ with $m_C(x) = max(m_A(x), m_B(x))$.
(4) The complement of a set A, denoted by A^C, is given by $m_{A^C}(x) = 1 - m_A(x)$.
(5) The size of set $|A|$ is given by $|A| = \sum m_A(x)$.

Remark: In these definitions, we identified fuzzy sets with their characteristic functions. There is a related geometrical interpretation developed by Kosko. Assume a finite set X with n elements. Now construct a n-dimensional hypercube (the Kosko cube). Its corners have coordinates $(x_1, \ldots, x_n), x_i \in \{0,1\}$. Consequently, they can be understood as representatives of the classical subsets of X: if $x_i = 1$ the subset represented by the corner contains x_i. Fuzzy set theory now extends set theory to the whole cube: each point of its interior represents a fuzzy subset of X by identifying its coordinates with the values of a characteristic function of a fuzzy subset. This geometrical point of view helps to visualize set operations (such as Venn diagrams do it for classical logic) but also nicely exhibit non-classical aspects of fuzzy logic. To give an example: For the set K represented by the center of the Kosko cube, it holds $K = K^C$, it is identical to its own complement.

Equipped with fuzzy set operations, we obtain the fuzzy ART algorithm (see [**Carpenter et al., 1991b**]) by substituting key operations of ART1 by fuzzy operations according to the table given below. We use the same notation as in the

previous sections and understand vectors as (ordered) sets: Consequently, classical set intersection is equivalent to the AND operation.

ART1	Fuzzy ART
Formation of hypothesis by maximizing T_i	
$T_i = \frac{\lvert\xi_j \cap w_i\rvert}{\epsilon + \lvert w_i\rvert}$	$T_i = \frac{\lvert\xi_j \bigwedge w_i\rvert}{\epsilon + \lvert w_i\rvert}$
Test of hypothesis	
$\frac{\lvert\xi_j \cap w_i\rvert}{\lvert\xi_j\rvert} \geq \rho$	$\frac{\lvert\xi_j \bigwedge w_i\rvert}{\lvert\xi_j\rvert} \geq \rho$
Fast learning	
$w_i^{\text{new}} = \xi \cap w_i^{\text{old}}$	$w_i^{\text{new}} = \xi \bigwedge w_i^{\text{old}}$

Besides extending ART1 to continuous values, fuzzy ART allows solving a further problem of ART1. The AND-operation in ART1 is irreversible and the values of the weights w_{ij} are restricted (at least in the algorithm, the learning in a network implementation requires at least temporarily non-binary values due to changes following a differential equation) to $\{0, 1\}$. This implies that noisy inputs can lead to degradation of the representatives of categories over time. This difficulty can be (at least partially) overcome by a "fast-commit-slow-recode" learning scheme, given by:

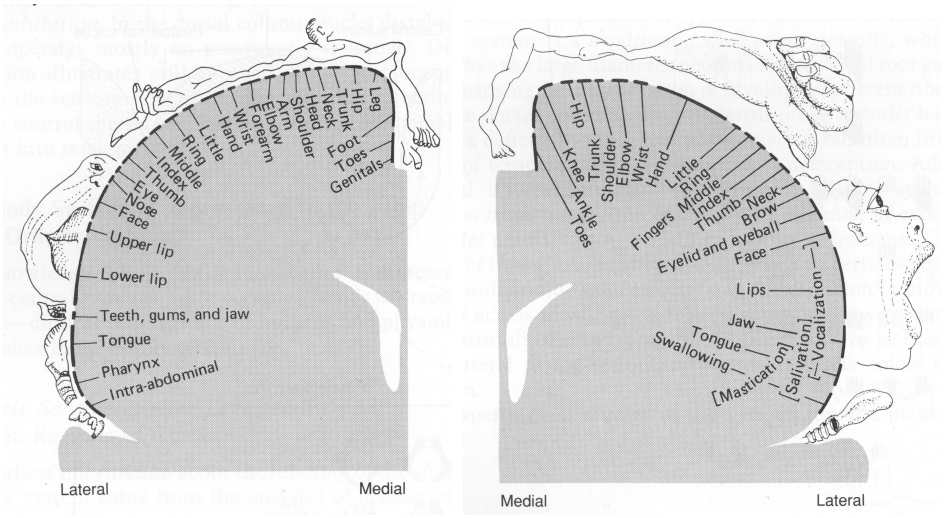
$$w_i^{\text{new}} = \beta(\xi \bigwedge w_i^{\text{old}}) + (1 - \beta) w_i^{\text{old}}. \tag{82}$$

Usually, the parameter β is set to one as long as w_i is uncommitted, and set to $\beta < 1$ afterwards.

The fuzzy ART algorithm as it is presented here suffers from a fundamental flaw, namely the "proliferation of categories". The problem occurs because the MIN-operator of fuzzy logic leads to a monotonic shift of the representatives of categories towards the origin $\{0, \ldots, 0\}$. As shown very instructively in an article by Grossberg and Carpenter [**Carpenter et al., 1991b**], the volume of the region in the Kosko cube which leads to the acceptance of a hypothesis represented by w_i becomes the smaller the closer w_i is situated to the origin. This means that in a situation where an infinite and randomly distributed flow of inputs is sent to a fuzzy ART system, a process as follows can happen repeatedly:

(1) A random input ξ_{j^*} is close to $\{1, \ldots, 1\}$.
(2) Assume that there is no w_i such that the vigilance criterion can be satisfied. An uncommitted $w_{i^*}^{(1)}$ has to be committed.
(3) Other random inputs may match with w_{i^*}. They shift $w_{i^*}^{(1)}$ towards the origin. The resulting series of category representatives is denoted by $w_{i^*}^{(n)}$.
(4) Although ξ_{j^*} and $w_{i^*}^{(1)}$ satisfy the vigilance criterion, this may not be anymore the case for ξ_{j^*} and some subsequent $w_{i^*}^{(n)}$. Consequently, a later presentation of ξ_{j^*} requires again the formation of a new category.

In [**Carpenter et al., 1991b**], Grossberg and Carpenter presented with the so called "complement coding" a solution to this problem. They showed that the fuzzy ART algorithm can be stabilized simply by employing a duplicated version of the input: Instead of a N-component vector $\xi = (\xi_1, \ldots, \xi_N)$, one uses a 2N-component vector $\hat{\xi} = (\xi, \xi^C) = (\xi_1, \ldots, \xi_N, 1 - \xi_1, \ldots, 1 - \xi_N)$.



(a) Sensory homunculus (b) Motor homunculus

FIGURE 6. A classical Penfield map. Somatic sensory and motor projections from and to the body surface and muscle are arranged in the cortex in somatotopic order [**Kandel et al., 1991**].

3. Feature mapping

There is a vast body of neurobiological evidence that the sensory and motor systems are represented as topographic maps in somatosensory cortex. Topographic maps have essentially three properties: (1) the input space, i.e. the space spanned by the patterns of sensory stimulation, is represented at the neural level (in other words, the input space is approximated by the neural system); (2) neighboring sensors are mapped onto nearby neural representations (a characteristic called *topology preservation* or *topological ordering*); (3) a larger number of neurons are allocated to regions on the body with high density of sensors (a property called *density matching*). A similar argument holds for the motor system. Figure 6 shows the classical Penfield map. It can be clearly seen that the two conditions, topology-preservation, and preservation of density hold for natural systems. For example, many neurons are allocated to the mouth and hand regions which correspond to extremely complex sensory and motor systems. The articulatory tract constitutes one of the most complex known motor systems, and the lip/tongue region is characterized by very high density of different types of sensors (touch, temperature, taste).

Now what could be the advantage of such an arrangement? It is clear that the potential of the sensory and the motor systems can only be exploited if there is sufficient neural substrate to make use of it, as stated in the principle of ecological balance. Also, according to the redundancy principle, there should be partially overlapping functionality in the sensory and motor systems. If there is topology preservation neighboring regions on the body are mapped to neighboring regions in the brain and if neighboring neurons have similar functionality (i.e. they have

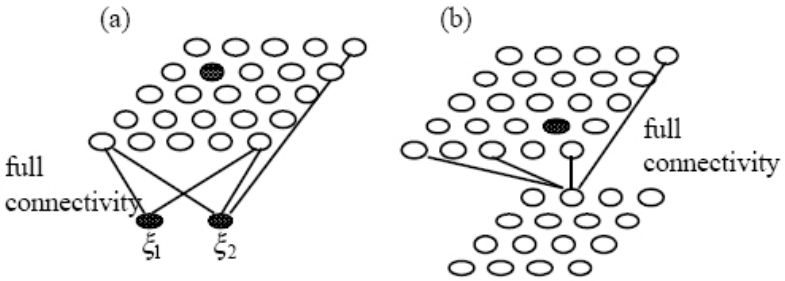


FIGURE 7. Two types of feature mapping networks. (a) the standard type with continuous-valued inputs ξ_1, ξ_2. (b) a biologically inspired mapping, e.g. from retina to cortex. Layers are fully connected, though only a few connections are shown (after [**Hertz et al., 1991**], p. 233).

learned similar things), if some neurons are destroyed, adjacent ones can partially take over, and if some sensors malfunction, neighboring cells can take over. Thus, topographic maps increase the adaptivity of an organism. So, it would certainly be useful to be able to reproduce some of this functionality in a robot. We will come back to this point later in the chapter.

So far we have devoted very little attention to the geometric arrangement of the competitive units. However, it would be of interest to have an arrangement such that nearby input vectors would also lead to nearby output vectors. Let us call the locations of the winning output units r^1, r^2 for the two input vectors ξ^1, ξ^2. If for $\xi^1 \to \xi^2, r^1 \to r^2$, and $r^1 = r^2$ only if ξ^1 and ξ^2 are similar, we call this a *feature map*. What we are interested in is a so-called *topology-preserving feature map*, also called a topographic map. But what we want is a mapping that not only preserves the neighborhood relationships but also represents the entire input space. There are two ways to do this, one exploiting the network dynamics, the other one being purely algorithmic.

3.1. Network dynamics with "Mexican Hat". We can use normal competitive learning but the connections are not strictly winner-take-all and they have the form of a so-called "Mexican hat". Remember that in a winner-take-all or in a Hopfield network all nodes are connected to all other nodes, which is no longer the case here. This is shown in figure 8. In a winner-take-all or in a Hopfield network all nodes are connected to all other nodes. There is no notion of neighborhood, i.e. all nodes are equally far from each other. Here, the nodes are on a grid (frequently, but not necessarily, two-dimensional) and thus there is a notion of distance which can in turn be used to define a topology. Note that this differs fundamentally from the networks we have studied so far.

Note that this network is no longer strictly "winner-take-all", but several nodes are active, given a particular input. Through learning, neighboring units learn similar reactions to input patterns. Nodes that are further away become sensitive to different patterns. In this sense, the network has a dimensionality, which is different from all the networks discussed in previous chapters. This alternative is computationally intensive: there are many connections, and there are a number of

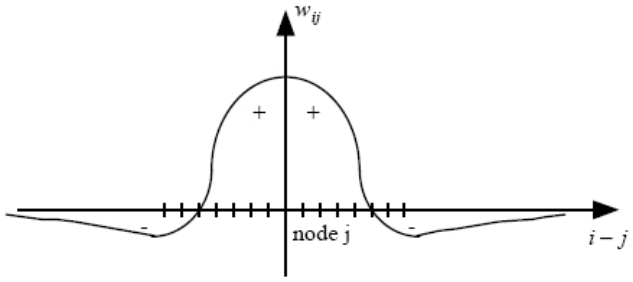


FIGURE 8. The "Mexican hat" function. The connections near node j are excitatory (positive), the ones a bit further away inhibitory (negative), and the ones yet further away, zero.

iterations required before a stable state is reached. Kohonen suggested a short-cut that has precisely the same effect.

3.2. Kohonen's Algorithm. In order to determine the winner, equation (73) is used, just as in the case of the standard competitive learning algorithm. What is different now is the learning rule

$$\Delta w_{ij} = \eta \Lambda(i, i^*)(\xi_j - w_{ij})(\forall i, \forall j) \tag{83}$$

where η is the learning rate, and $\Lambda(i, i^*)$ is a neighborhood function. It is typically chosen such that it is 1 at the winning node i^* and drops off with increasing distance. This is captured in formula (84).

$$\Lambda(i, i^*) = \begin{cases} 1 & \text{for } i = i^* \\ \text{drops off with distance} & |r_i - r_{i^*}| \end{cases} \tag{84}$$

Note that near $i^* \Lambda(i, i^*)$ is large which implies that the weights are changed strongly in similar ways as the ones of i^*. Further away, the changes are only minor. One could say that the topological information is captured in $\Lambda(i, i^*)$. The learning rule can verbally be described as follows: w_{ij} is pulled into the direction of the input, ξ (just as in the case of standard competitive learning). The processing of neighboring units is such that they learn similar things as their neighbors. This is called an "elastic net".
In order for the algorithm to converge, $\Lambda(i, i^*)$ and η have to change dynamically: we start with a large region and a large learning rate η and they are successively reduced. A typical choice of $\Lambda(i, i^*)$ is a Gaussian:

$$\Lambda(i, i^*) = \exp(\frac{-|r_i - r_{i^*}|^2}{2\sigma^2}) \tag{85}$$

where σ is the "width" which is successively reduced over time. The time dependence of $\eta(t)$ and $\sigma(t)$ can take various form, e.g. $\frac{1}{t}$, or $\eta(t) \propto t^{-\alpha}, 0 < \alpha \leq 1$. The development of the Kohonen algorithm can be visualized as in figure 10 (another way of visualizing Kohonen maps is shown in section 3.4). Figure 9 shows how a network "unfolds": it can be seen how the weight vectors which are

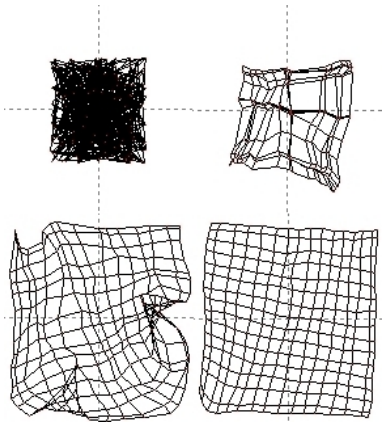


FIGURE 9. "Unfolding" of a Kohonen network map

initialized to be in the center, gradually start covering the entire input space (only two dimensions shown).

Although this Gaussian neighborhood function looks very similar to the "Mexican hat" function, the two are not to be confounded. The "Mexian hat" is about connection strengths and thus influences the network dynamics, the neighborhood function is only used in the learning rule and indicates how much the neighboring nodes to the winner learn. The advantage of the Kohonen algorithm is that we do not have to deal with the complex network dynamics but the results are the same. Note also, that the "Mexican hat" has negative regions which is necessary to get stable "activation bubbles" (instead of a single active node in the case of standard competitive learning), whereas the Gaussian is always positive.

3.3. Properties of Kohonen maps. Kohonen makes a distinction between two phases, a self-organizing or an ordering phase, and a convergence phase. In this phase (up to a 1000 iterations) the topological ordering of the weight vectors takes place. In the convergence phase the feature map is fine-tuned to provide statistical quantification of the input space ([**Haykin, 1994**], p. 452).

As we said at the beginning of our discussion of feature maps, they should (1) approximate the input space, (2) preserve neighborhood relationships, and (3) match densities. Do Kohonen maps (or SOMs) have these desirable characteristics as well? It can be shown theoretically and has been demonstrated in simulations (e.g. [**Haykin, 1994**], p. 461) that for Kohonen maps (1) and (2) hold, but in fact tend to somewhat overrepresent regions of low input density and to underrepresent regions of high input density. In other words, the Kohonen algorithms fails to provide an entirely faithful representation of the probability distribution that underlies the input data.

However, the inpact of this problem can be alleviated by adding a "conscience" to competitive learning, as suggested by [**DeSieno et al., 1988**]. He introduced a form of memory to track the cumulative activity of individual neurons in the Kohonen layer by introducing a bias against those nodes that often win the competition.

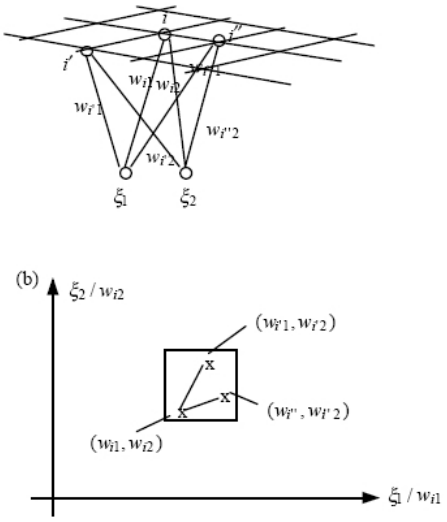


FIGURE 10. Visualization of the Kohonen algorithm. In this example, the weights are initialized to a square in the center. The points in the x-y plane represent the weights that eventually approximate the input space. The links connect neighboring nodes.

In other words, the effect of the algorithm is that each node, irrespective of position on the grid, has a certain probability of being activated. This way, better density matching can be achieved.

3.4. Interpretation of Results. There is an additional problem that we need to deal with when using Kohonen maps. Assuming that the map has converged to some solution, the question then arises: "*now what?*" So, we do have a result, but we also need to do something with it. Typically it is a human being that simply "looks" at the results and provides an interpretation. For example, in Kohonen's "Neural" Phonetic Typewriter, the nodes in the output layer (the Kohonen layer) correspond to phonemes (or rather quasi-phonemes, because they have a shorter duration), and one can easily detect that related phonemes are next to each other (e.g. [**Kohonen, 1988**]).

[**Ritter and Kohonen, 1989**] introduced the concept of contextual maps or semantic maps. In this approach, the nodes are labeled with the test pattern that excites this neuron maximally. This procedure resembles the experimental procedure in which sites in a brain are labeled by those stimulus features that are most effective in exciting neurons at this site. The labeling produces a partitioning of the Kohonen layer into a number of coherent regions, each of which contains neurons that are specialized for the same pattern. In the example of figure 11, each training pattern was a coarse description of one of 16 animals (using a data vector of 13 simple binary-valued features – e.g. small medium, big, 2 legs, 4 legs, hair, hooves, feathers, likes-to-fly, run, swim, etc.). Evidently, in this case a topographic map that exhibits the similarity relationships among the 16 animals has been formed,

and we get birds, peaceful species, and hunters.

Another way to deal with the question "*now what?*" is of interest especially when using self-organizing maps for robot control. SOMs can be extended and combined with a supervised component through which the robot can learn what to do, given a particular classification. This is described in section 4, below.

3.5. A Note on Dimensionality. One of the reasons Kohonen maps have become so popular is because they perform a dimensionality reduction – reducing a high-dimensional space to just a few dimensions, while preserving the topology. In the literature, often two-dimensional Kohonen maps are discussed, i.e. maps where the Kohonen or output layer constitutes a two-dimensional grid, and thus the neighborhood function is also two-dimensional (a bell-shaped curve). It should be noted, however, that this dimensionality reduction only works, i.e. is only topology-preserving if the data are indeed two-dimensional. You can think of a 3-D space where two points are far apart, but in the projection (as one example) on the x-y plane, they are very close together (see also the discussion on the properties of topographic maps at the beginning of the section).

4. Extended feature maps - robot control

Kohonen maps can be used for clustering, as we have seen. But they are also well-suited for controlling systems that have to interact with the real world, e.g. robots: given particular input data, what action should the system take? For this purpose, there is an extended version of the Kohonen maps. They work as follows (see figure 12):

(1) Given an input-output pair $(\underline{\xi}, u)$ where u is the desired output (i.e. the action, e.g. the angles of a robot arm - normally a vector).
(2) Determine the winner i^*
(3) Execute a learning step (as we have done so far):
$\Delta w_{i^*}^{(in)} = \eta \Lambda_{i^*} \cdot (\underline{\xi} - w_{i^*}^{(in)})$

dog	dog	fox	fox	fox	cat	cat	cat	eagle	eagle
dog	dog	fox	fox	fox	cat	cat	cat	eagle	eagle
wolf	wolf	wolf	fox	cat	tiger	tiger	tiger	owl	owl
wolf	wolf	lion	lion	lion	tiger	tiger	tiger	hawk	hawk
wolf	wolf	lion	lion	lion	tiger	tiger	tiger	hawk	hawk
wolf	wolf	lion	lion	lion	owl	dove	hawk	dove	dove
horse	horse	lion	lion	lion	dove	hen	hen	dove	dove
horse	horse	zebra	cow	cow	cow	hen	hen	dove	dove
zebra	zebra	zebra	cow	cow	cow	hen	hen	duck	goose
zebra	zebra	zebra	cow	cow	cow	duck	duck	duck	goose

FIGURE 11. Semantic map obtained through the use of simulated electrode penetration mapping. The map is divided into three regions representing: birds, peaceful species, and hunters. [**Haykin, 1994**]

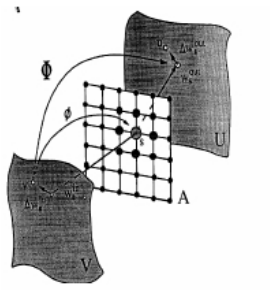


FIGURE 12. Illustration of the extended Kohonen algorithm.

Applying steps (1), (2) and (3) would lead to clusters as we had them previously. But now, we want to –in addition– exploit this result to control a robot in a desirable way. This is done by adding an additional step:

(4) Execute a learning step for the output weights:
$\Delta w_{i^*}^{(out)} = \eta' \Lambda'_{i^*} \cdot (u - w_{i^*}^{(out)})$

This is the *supervised* version of the extended Kohonen algorithm. For example, instead of providing the exact values for the output as in the supervised algorithm it is possible to provide only a global evaluation function, a reinforcement value. The latter requires a component of either random exploration or "directed exploration", depending on the problem situation. These algorithms are called *reinforcement learning algorithms*. We will not discuss them here.

4.1. Example: The adaptive light compass [Lambrinos, 1995]. Desert ants can use the polarized light from the sky for navigation purposes, as a kind of celestial compass. As they leave the nest, which is essentially a hole in the ground, the ants turn around their own axes. The purpose of this activity seems to be to "calibrate" the compass. Dimitri Lambrinos, a few years back, tried to reproduce some of these mechanisms on a simple Khepera robot with 8 IR sensors and two motors, as you know it from the Artificial Intelligence course. The input consists of the IR sensor readings, the outputs of the motor speeds. In this case, the IR sensors are used as light sensors. Then the procedure 1 to 4 is applied. The input-desired output pair is constructed as follows.
The input $\underline{\xi}$ is calculated from the normalized sensory signals from the 8 light sensors. The corresponding desired output, here called $\underline{o}$, is calculated from the wheel encoder (the sensors that measure how much the wheels turn). In order to find the desired output, the robot performs a "calibration run" which consists, in essence, of a turn of 360 degrees around its own axis.

During this process the robot collects the input data from the light sensors, and associates each of these pattern with the angle of how much the robot would have to turn to get back into the original direction, i.e. the desired output for this particular light pattern, the input pattern (see figure 13). The connections from the input layer (the sensors) to the map layer are trained with the standard Kohonen learning algorithm. The network architecture is shown in figure 14. The output

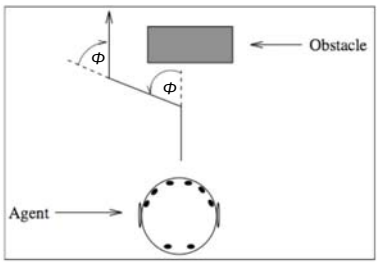


FIGURE 13. Obstacle avoidance and turning back using the adaptive light compass.

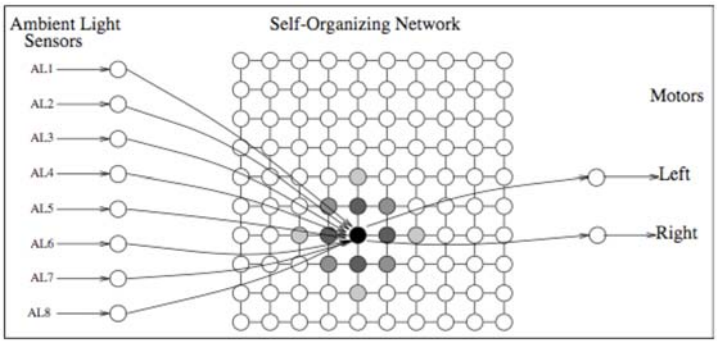


FIGURE 14. Network of an adaptive light compass.

weights $\underline{o} = w_{i*}^{(out)}$, i.e. the weights from the map layer to the motor neurons, are trained using the formula of step 4 in the extended feature map:

$$\underline{o} = (o_l, o_r) = \left(\frac{\phi}{\pi} - 0.5, -\frac{\phi}{\pi} + 0.5\right) \tag{86}$$

where o_l,o_r is the speed of the left and right wheel respectively, normalized in the range $[-1, 1]$ (-1 stands for the maximum speed backwards, 1 stands for the maximum speed forward).

The learning proceeds according to the standard extended Kohonen algorithm, i.e.

(1) find winner i^*
(2) update weights:

$$\begin{aligned}
\Delta\underline{w}_i^{(in)} &= \eta\Lambda(i, i^*) \cdot (\underline{\xi} - \underline{w}_i^{(in)}) \\
\Delta\underline{w}_i^{(out)} &= \eta'\Lambda'(i, i^*) \cdot (\underline{o} - \underline{w}_i^{(out)}) \\
\Lambda(i, i^*) &= e^{-\frac{|i-i^*|}{2\sigma^2(t)}}
\end{aligned}$$

where $\sigma(t)$ determines the diameter of the neighborhood where changes in the weights are going to occur and is defined as

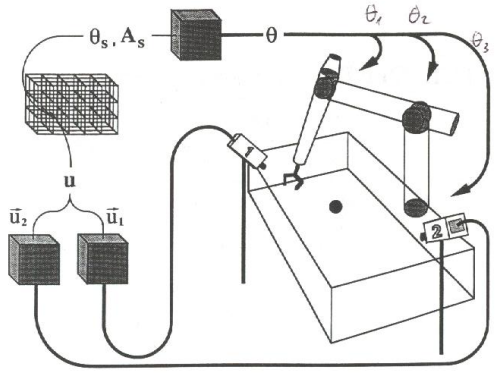


FIGURE 15. Schematic representation of the positioning process: $\underline{u_1} = (u_1^1, u_1^2), (u_2^1, u_2^2)$, position of the target (dot in the robot arm's workspace) in camera$_1$ and camera$_2$ which are passed on to the 3-dimensional Kohonen network. The winner neuron i^* has two output values $\underline{\theta_{i^*} = (\theta_1, \theta_2, \theta_3)}$ and $\underline{A}_{i^*}$, the so-called Jacobian matrix, which are both needed to determine the joint angles of the robot arm in order to move towards the target (the dot). [**Ritter et al., 1991**]

$$\sigma(t) = 1.0 + (\sigma_{max} - 1.0) \cdot \frac{t_{max} - t}{t_{max}} \tag{87}$$

where σ_{max} is the initial value of $\sigma(t)$ and t_{max} is the maximum number of training steps. With this setting, $\sigma(t)$ decreases over time to 1. The learning rate is also defined as a function of time as follows:

$$\eta(t) = \eta_{max} \cdot \frac{t_{max} - t}{t_{max}} \tag{88}$$

A similar procedure can be applied to the control of a robot arm through input from 2 cameras (see figure 15). How the Kohonen-map evolves is shown in figure 16. While initially (top row) the distribution of the weight vectors (from camera to Kohonen network) is random, they start, over time, focusing on the relevant subdomain.

The resulting robot controller turns out to be surprisingly robust and works reliably even if there is a lot of noise in the sensory input and the light patterns are fuzzy and hard to distinguish from one another (for further details, see [**Lambrinos, 1995**].

5. Feature Extraction by Principle Component Analysis (PCA)

In unsupervised learning, the target outputs of our network are not available. A desired input-output mapping cannot be learned by our network, because there is no supervisor who tells us how the output should look like for a given input. Instead, we can try to extract useful information from the input data itself, by looking at its statistical properties. This can help us to identify interesting features, which we

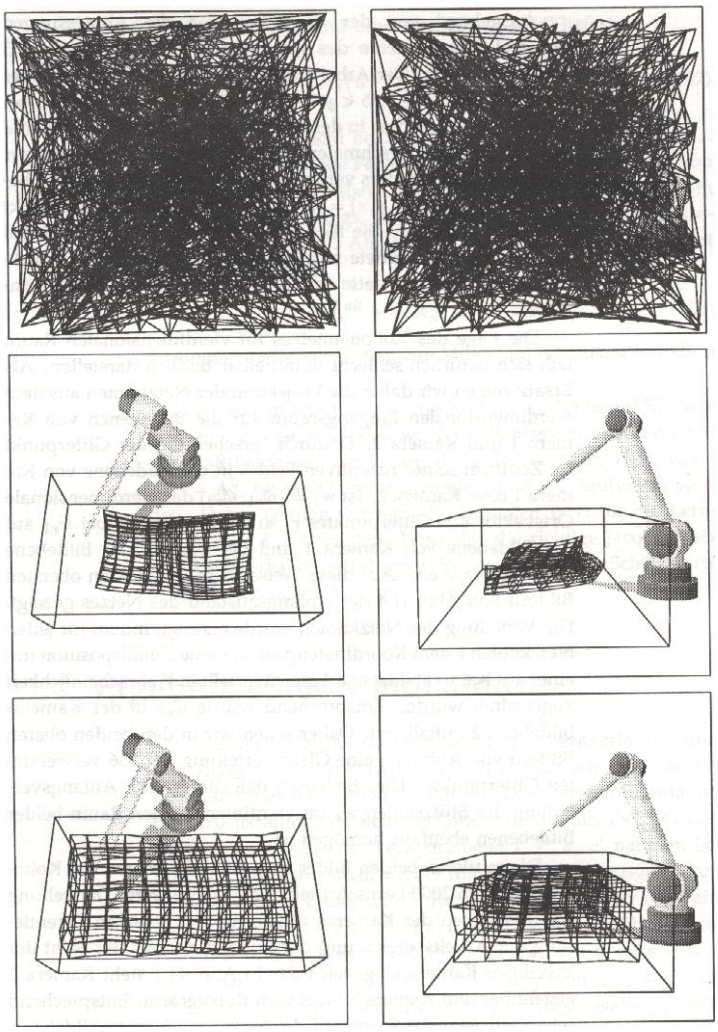


FIGURE 16. Position of Kohonen network at the beginning (top row), after 2000 steps (middle row), and after 6000 steps. [**Ritter et al., 1991**]

then can extract from the input data and discard all the non-informative parts, a process called *feature extraction* or *dimensionality reduction*. A prominent method which does this is the Principle Component Analysis (PCA) or Karhunen-Loève Transform.

Imagine you are given a high dimensional data set (say $d = 10000$). Maybe it is not a good idea to build your network with 10000 input nodes, but instead only select the interesting dimensions as input nodes. As an example look at the two dimensional data set in figure 17 (a). Which of the two axes would you want to keep and which would you discard in order to reduce the dimensionality? Intuitively you would choose ξ_1, because the data is more spread along this axis. We can say, ξ_1 contains more information than ξ_2, because it has a higher *variance* and if we

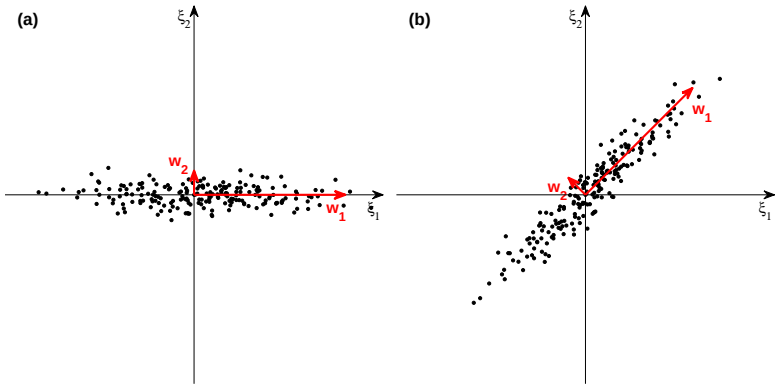


FIGURE 17. Principle Component Analysis. The first principle component w_1 points into the direction of maximum variance. The second principle component w_2 is orthogonal to the first one.

discard ξ_2 we lose only little information about the data. Now look at the data set in figure 17 (b). Both axes have the same variance, so obviously no axis is better than the other to keep or discard. Instead of choosing one of the axes, we can just choose one *direction* $\underline{w}_1$ of the input space, in which the variance is the largest. Then we project all the data onto this direction and use the projected data $\underline{\xi}^T \underline{w}_1$ as an input to our network, while still preserving most of the information of the original data. But how can we find the direction of maximum variance? Remember that the covariance matrix is defined as

$$Cov[\underline{\xi}] = \frac{1}{N} \sum_{\mu=1}^{N} (\underline{\xi}^{(\mu)} - \bar{\underline{\xi}})(\underline{\xi}^{(\mu)} - \bar{\underline{\xi}})^T \tag{89}$$

where $\bar{\underline{\xi}}$ is the mean. The diagonal of $Cov[\underline{\xi}]$ contains the variances of the individual dimensions and the non-diagonal cells contain the covariances between dimensions. Moreover, if we have zero-mean data (which can be easily achieved by subtracting the mean from all patterns), the covariance matrix is equal to the *correlation matrix*:

$$C = \frac{1}{N} \sum_{\mu=1}^{N} \underline{\xi}^{(\mu)} \underline{\xi}^{(\mu)T} \tag{90}$$

The variance of the projected data $\underline{\xi}^T \underline{w}$ is then given by $\underline{w}^T C \underline{w}$. Our goal is now to find the $\underline{w}$ which maximizes this expression. Because we are only interested in the direction of $\underline{w}$, but not in its length, we choose it to have unit length $\|\underline{w}\|^2 = 1$. This forms an optimization problem with constraint, which can be solved using Lagrange multipliers:

$$L(\underline{w}, C, \lambda) = \underline{w}^T C \underline{w} + \lambda(1 - \underline{w}^T \underline{w}) \tag{91}$$

By setting its derivative with respect to $\underline{w}$ to 0, we get:

$$\frac{\partial L}{\partial \underline{w}} = 0 = C\underline{w} - \lambda \underline{w}$$

$$C\underline{w} = \lambda \underline{w} \tag{92}$$

This is an eigenvalue problem and we see that the $\underline{w}$ we are looking for is an eigenvector of the correlation matrix C. Moreover, we see that the eigenvalue corresponding to $\underline{w}$ is equal to the variance in this direction $\lambda = \underline{w}^T C \underline{w}$, where we made use of the constraint $\underline{w}^T \underline{w} = 1$. This is good news, because all we have to do for getting desired direction vector is to chose the eigenvector $\underline{w}$ which corresponds to the highest eigenvalue and project the input data onto it.

The direction of maximum variance is called the first *principle component* of the data. We can extend this idea by finding a second direction orthogonal to the first one, which contains the second largest variance and is called the second principle component. Then the third one etc. We have as many components as we have input dimensions and the projection into the principle component space is equivalent to a coordinate transform. We can write the set of principle components in one matrix and order them with decreasing variance $\lambda_1 > \lambda_2 > \cdots > \lambda_d \Rightarrow W = [\underline{w}_1, \underline{w}_2, \cdots, \underline{w}_d]$. To project a pattern $\underline{\xi}$ into the principle component space, we simply multiply it by W:

$$\underline{\tilde{\xi}} = W^T \underline{\xi} = [\underline{\xi}^T \underline{w}_1, \underline{\xi}^T \underline{w}_2, \ldots, \underline{\xi}^T \underline{w}_d] \tag{93}$$

To extract features of the data or to reduce the dimensionality ($\mathbb{R}^d \to \mathbb{R}^l$) of the data with minimal loss of information, we can choose the first $l < d$ principle components only $W_l = [\underline{w}_1, \underline{w}_2, \cdots, \underline{w}_l]$

$$\underline{\tilde{\xi}} = W_l^T \underline{\xi} = [\underline{\xi}^T \underline{w}_1, \underline{\xi}^T \underline{w}_2, \ldots, \underline{\xi}^T \underline{w}_l] \tag{94}$$

We call $\underline{\tilde{\xi}}$ a *feature vector* and can use it instead of $\underline{\xi}$ as input for our network. In our example with $d = 10000$ dimensions of $\underline{\xi}$, we may choose a feature vector $\underline{\tilde{\xi}}$ of $l = 100$ dimensions and thereby significantly reduce the weight space of our network, while preserving most of the important information.

6. Hebbian learning

6.1. Introduction: plain Hebbian learning and Oja's rule. Just as in the case of feature mapping, there is no teacher in Hebbian learning but the latter does not have the "competitive" character of feature mapping. Let us look at an example, Hebbian learning with one output unit.

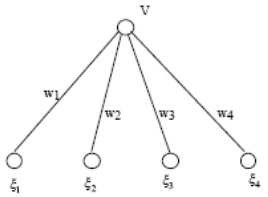


FIGURE 18. Unsupervised Hebbian learning. The output unit is linear.

The output unit is linear, i.e.

$$V = \sum_j w_j \xi_j = \underline{w}^T \underline{\xi} \tag{95}$$

The vector components ξ_j are drawn from a probability distribution $P(\xi)$. At each time step a vector $\underline{\xi}$ is applied to the network. With Hebbian learning, the network learns, over time, how "typical" a certain $\underline{\xi}$ is for the distribution $P(\xi)$: the higher the probability of $\underline{\xi}$, the larger the output V on average.
Plain Hebbian learning:

$$\Delta w_j = \eta V \xi_j \tag{96}$$

We can clearly see that frequent patterns have a bigger influence on the weights than infrequent ones. However, the problem is that the weights keep growing. Thus, we have to ensure that the weights are normalized. This can be automatically achieved by using Oja's rule:

$$\Delta w_j = \eta V(\xi_j - V w_j) \tag{97}$$

It can be shown that with this rule, the weight vector converges to a constant length, $\|\underline{w}\|^2 = 1$. How Oja's rule works is illustrated in figure 19. The thin line indicates the "path" of the weight vector.

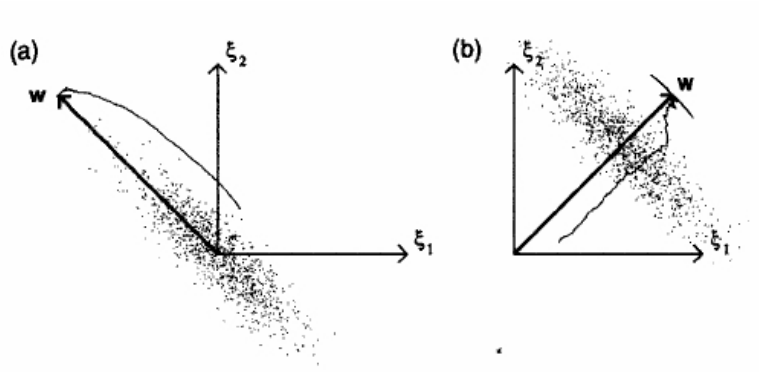


FIGURE 19. Illustration of Oja's rule (from [**Hertz et al., 1991**], p. 201).

Because we have linear units, the output V is just the component of the input vector $\underline{\xi}$ in the direction of $\underline{w}$. In other words, Oja's rule chooses the direction of $\underline{w}$ to maximize $\langle V^2 \rangle$. For distributions with zero mean (case (a) in figure 19), this corresponds to variance maximization at the output and thus to finding the first principal component. We can now extend this method to find the first l principle components, by using a layer of l linear units and connecting them in a way, that each unit i receives the input $\underline{\xi}$ minus the weighted output of all preceding units $1, \ldots, i-1$

$$\Delta w_{ij} = \eta V_i \left[\left(\xi_j - \sum_{k=1}^{i-1} V_k w_{kj} \right) - V_i w_{ij} \right]$$

or shortened:

$$\Delta w_{ij} = \eta V_i \left(\xi_j - \sum_{k=1}^{i} V_k w_{kj} \right) \tag{98}$$

This is known as Sanger's rule or *Generalized Hebbian Learning*. Figure 20 illustrates how such a network would look like. For more details, consult [**Hertz et al., 1991**], pp. 201-204.

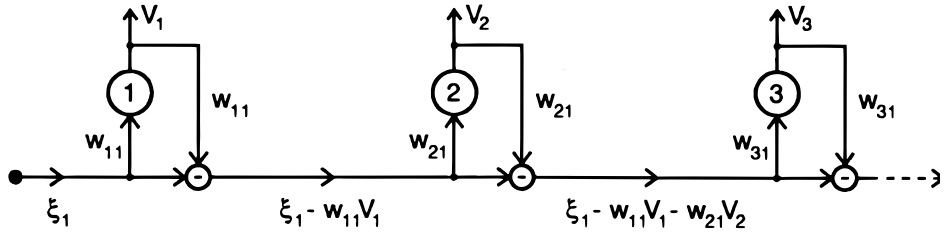


FIGURE 20. Network implementing Sanger's rule (from [**Hertz et al., 1991**], p. 208). For simplicity only one input line is shown (ξ_1). The weighted output of each unit i is subtracted from the input before it is fed into the next unit $i+1$.

6.2. An application of Hebbian learning to robots: distributed adaptive control. (from [**Pfeifer and Scheier, 1999**]) Figure 21 shows an example of a simple robot.

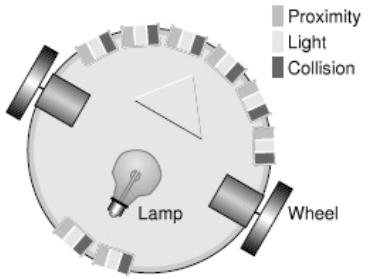


FIGURE 21. The generic robot architecture.

There is a ring of proximity sensors. They yield a measure of "nearness" to an obstacle: the nearer, the higher the activation. If the obstacle is far away, activation will be zero (except for noise). There are also a number of collision sensors, which transmit a signal when the robot hits an obstacle. In addition, there are light sensors on each side. The wheels are individually driven by electrical motors. If both motors turn at equal speeds, the robot moves straight, if the right wheel is stopped and only the left one moves, the robot turns right, and if the left wheel moves forward and the right one backward with equal speed, the robot turns on the spot. This architecture represents a generic low-level ontology for a particular simple class of robots. It is used to implement the "Distributed Adaptive Control" architecture.

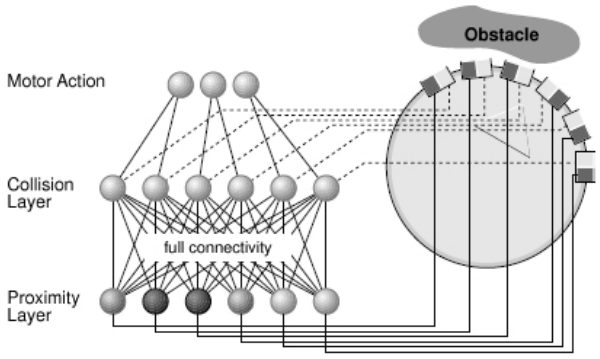


FIGURE 22. Embedding of the network in the robot.

6.3. Network architecture and agent behavior. We now define the control architecture. The robot has a number of built-in reflexes. If it hits with a collision sensor on the right, it will back up just a little bit and turn to the left (and vice versa). There is another reflex: Whenever the robot is sensing light on one side, it will turn towards that side. If there are no obstacles and no lights it will simply move forward. How can we control this robot with a neural network? Figure 22 shows how a neural network can be embedded in the robot. For reasons of simplicity, we omit the light sensors for the moment. Note also that the proximity sensors are distributed over the front half of the robot in this particular architecture. Each sensor is connected to a node in the neural network: the collision sensors to nodes in the collision layer, the proximity sensors to nodes in the proximity layer. The collision nodes are binary threshold, i.e., if their summed input hi is above a certain threshold, their activation value is set to 1, otherwise it is 0. Proximity nodes are continuous. Their value depends on the strength of the signal they get from the sensor. In figure 22, it can be seen that the nodes in the proximity layer show a certain level of activation (the stronger the shading of a circle, the stronger its activation), while the nodes in the collision layer are inactive (0 activation) because the robot is not hitting anything (i.e., none of the collision sensors is turned on).

The proximity layer is fully connected to the collision layer in one direction (the arrows are omitted because the figure would be overloaded otherwise). If there are six nodes in the proximity layer and 5 in the collision layer as in figure 22 there are 30 connections. The nodes in the collision layer are connected to a motor output layer. These connections implement the basic reflexes. We should mention that the story is exactly analogous for the light sensors, so there is also a layer of nodes (in this case only 2) for each light sensor.

In figure 22 the robot is moving straight and nothing happens. If it keeps moving then it will eventually hit an obstacle. When it hits an obstacle (figure 23) the corresponding node in the collision layer is turned on (i.e., its activation is set to 1). As there now is activation in a collision node and simultaneously in several proximity nodes, the corresponding connections between the proximity nodes and the active collision node are strengthened through Hebbian learning.

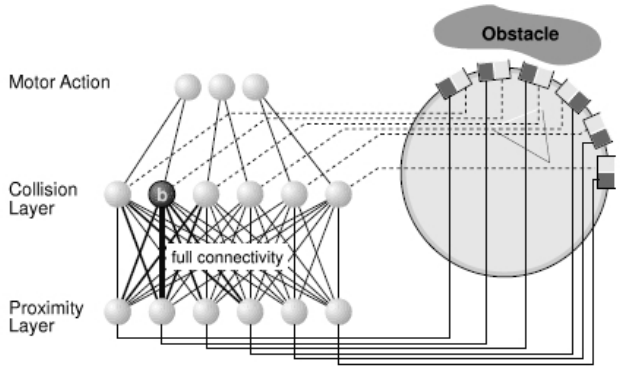


FIGURE 23. The robot hitting an obstacle.

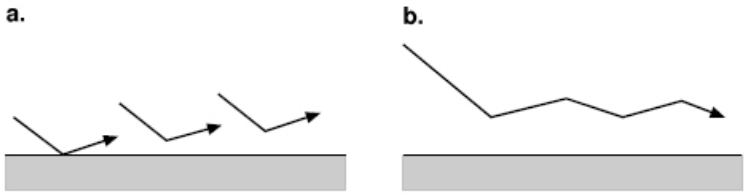


FIGURE 24. Development of robot behavior over time. (a) obstacle avoidance behavior, (b) wallfollowing behavior.

This means that next time around more activation from the proximity layer will be propagated to the collision layer. Assume now that the robot hits obstacles on the left several times. Every time it hits, the corresponding node in the collision layer becomes active and there is a pattern of activation in the proximity layer. The latter will be similar every time. Thus the same connections will be reinforced each time. Because the collision nodes are binary threshold, the activation originating from the proximity layer will at some point be strong enough to get the collision node above threshold without a collision. When this happens, the robot has learned to avoid obstacles. In the future it should no longer hit obstacles.

The robot continues to learn. Over time, it will start turning away from the object earlier. This is because two activation patterns in the proximity sensor, taken within a short time interval, are similar when the robot is moving towards an obstacle. Therefore, as the robot encounters more and more obstacles as it moves around it will continue to learn, even if it does no longer hit anything. The behavior change is illustrated in figure 24 (a).

Mathematically, the input to the node i in the collision layer can be written as follows:
$h_i = c_i + \sum_{j=1}^{N} w_{ij} \cdot p_j$
where p_j is the activation of node j in the proximity layer, c_i the activation resulting from the collision sensor, w_{ij} the weight between the proximity layer and the collision layer, and h_i the local field (the summed activation) at collision node i. c_i

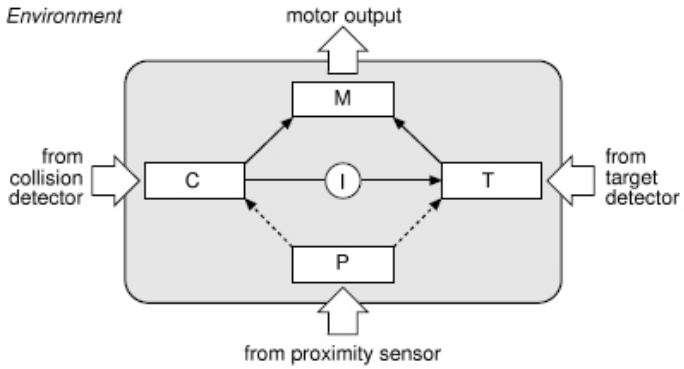


FIGURE 25. The complete "Distributed Adaptive Control" architecture.

is either 1 or 0, depending on whether there is a collision or not. p_j is a continuous value between 0 and 1, depending on the stimulation of the proximity sensor j (high stimulation entails a high value, and vice versa). N is the number of nodes in the proximity layer. Let us call a_i the activation of node i in the collision layer. Node i is - among others - responsible for the motor control of the agent. a_i is calculated from h_i by means of a threshold function g:

$$a_i = g(h_i) = \begin{cases} 0 & : h_i < \Theta \\ 1 & : h_i \geq \Theta \end{cases}$$

The weight change is as follows:

$$\Delta w_{ij} = \frac{1}{N}(\eta \cdot a_i \cdot p_j - \varepsilon \cdot \overline{a} \cdot w_{ij})$$

where η is the learning rate, N the number of units in the proximity layer as above, $\overline{a}$ the average activation in the collision layer, and ε the forgetting rate. Forgetting is required because otherwise the weights would become too large over time. Note that in the forgetting term we have $\overline{a}$, the average activation in the collision layer. This implies that forgetting only takes place when something is learned, i.e., when there is activation in the collision layer (see figure 24). This is also called *active forgetting*. The factor $\frac{1}{N}$ is used to normalize the weight change.

The complete "Distributed Adaptive Control" architecture is shown in figure 25. A target layer (T) has been added. Its operation is analogous to the collision layer (C). Assume that there are a number of light sources near the wall. As a result of its built-in reflex, the robot will turn towards a light source. As it gets close to the wall, it will turn away from the wall (because it has learned to avoid obstacles). Now the turn-toward-target reflex becomes active again, and the robot wiggles its way along the wall. This is illustrated in figure 24 (b). Whenever the robot is near the wall, it will get stimulation in the proximity layer. Over time, it will associate light with lateral stimulation in the proximity sensor (lateral meaning "on the side"). In other words, it will display the behavior of figure 24 (b) also if there is no longer a light source near the wall.

Bibliography

[Amit, 1989] Amit, D. (1989). *Modeling Brain Function: The World of Attractor Neural Networks*. Cambridge University Press.

[Anderson, 1995] Anderson, J. (1995). *An Introduction to Neural Networks*. Mit Press. (A well written introduction the field of neural networks by one of the leading experts of the field, especially geared towards the biologically interested reader.).

[Anderson and Rosenfeld, 1988] Anderson, J. and Rosenfeld, E. (1988). *Neurocomputing: Foundations of Research*. Cambridge, Mass. MIT Press.

[Beer, 1996] Beer, R. (1996). Toward the Evolution of Dynamical Neural Networks for Minimally Cognitive Behavior. *In P. Maes, M. Mataric, J.-A. Meyer, J. Pollack, and S.W. Wilson, eds. From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, pages 421–429.

[Beer, 2003] Beer, R. (2003). The Dynamics of Active Categorical Perception in an Evolved Model Agent. *Adaptive Behavior*, 11(4):209–243.

[Beer and Gallagher, 1992] Beer, R. and Gallagher, J. (1992). Evolving Dynamical Neural Networks for Adaptive Behavior. *Adaptive Behavior*, 1(1):91–122.

[Bennett and Campbell, 2000] Bennett, K. and Campbell, C. (2000). Support vector machines: hype or hallelujah? *ACM SIGKDD Explorations Newsletter*, 2(2):1–13.

[Bienenstock et al., 1982] Bienenstock, E., Cooper, L., and Munro, P. (1982). Theory for the development of neuron selectivity: orientation specificity and binocular interaction in visual cortex. *Journal of Neuroscience*, 2(1):32–48.

[Bryson and HO, 1969] Bryson, A. and HO, Y. (1969). Applied Optimal Control. *New York: Blaisdell*.

[Carpenter and Grossberg, 1990] Carpenter, G. and Grossberg, S. (1990). ART3: Hierarchical search using chemical transmitters in self-organizing pattern recognition architectures. *Neural Networks*, 3(2):129–152.

[Carpenter and Grossberg, 2003] Carpenter, G. and Grossberg, S. (2003). Adaptive resonance theory. the handbook of brain theory and neural networks.

[Carpenter et al., 1987] Carpenter, G., Grossberg, S., et al. (1987). ART 2: Self-organization of stable category recognition codes for analog input patterns. *Applied Optics*, 26(23):4919–4930.

[Carpenter et al., 1991a] Carpenter, G., Grossberg, S., and Rosen, D. (1991a). Art 2-A: an adaptive resonance algorithm for rapid category learning and recognition. *Neural Networks*, 4(4):493–504.

[Carpenter et al., 1991b] Carpenter, G., Grossberg, S., and Rosen, D. (1991b). Fuzzy ART: Fast stable learning and categorization of analog patterns by an adaptive resonance system. *Neural Networks*, 4(6):759–771.

[Cho, 1997] Cho, S. (1997). Neural-network classifiers for recognizing totally unconstrainedhandwritten numerals. *Neural Networks, IEEE Transactions on*, 8(1):43–53.

[Churchland and Sejnowski, 1992] Churchland, P. and Sejnowski, T. (1992). *The Computational Brain*. Bradford Books.

[Clark and Thornton, 1997] Clark, A. and Thornton, C. (1997). Trading spaces: Computation, representation, and the limits of uninformed learning. *Behavioral and Brain Sciences*, 20(01):57–66.

[DeSieno et al., 1988] DeSieno, D., Inc, H., and San Diego, C. (1988). Adding a conscience to competitive learning. *Neural Networks, 1988., IEEE International Conference on*, pages 117–124.

[Elman, 1990] Elman, J. (1990). Finding Structure in Time. *Cognitive Science*, 14:179–211.

[Fahlman and Lebiere, 1990] Fahlman, S. and Lebiere, C. (1990). The cascade-correlation learning architecture. *Advances in neural information processing systems II*, pages 524–532.

[Gomez et al., 2005] Gomez, G., Hernandez, A., Eggenberger Hotz, P., and Pfeifer, R. (2005). An adaptive learning mechanism for teaching a robot to grasp. *International Symposium on Adaptive Motion of Animals and Machines (AMAM 2005) Sept 25th-30th, Ilmenau, Germany.*

[Gorman et al., 1988a] Gorman, P. et al. (1988a). Analysis of hidden units in a layered network trained to classify sonar targets. *Neural Networks*, 1(1):75–89.

[Gorman et al., 1988b] Gorman, R., Sejnowski, T., Center, A., and Columbia, M. (1988b). Learned classification of sonar targets using a massively parallelnetwork. *Acoustics, Speech, and Signal Processing [see also IEEE Transactions on Signal Processing], IEEE Transactions on*, 36(7):1135–1140.

[Grossberg, 1987] Grossberg, S. (1987). Competitive Learning: From Interactive Activation to Adaptive Resonance. *Cognitive Science*, 11(1):23–63.

[Haykin, 1994] Haykin, S. (1994). *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[Hearst, 1998] Hearst, M. A. (1998). Support vector machines. *IEEE Intelligent Systems*, 13(4):18–28.

[Hertz et al., 1991] Hertz, J., Krogh, A., and Palmer, R. (1991). *Introduction to the Theory of Neural Computation*. Redwood City, CA: Addison-Wesley.

[Hopfield, 1982] Hopfield, J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proc. Natl. Acad. Sci. USA*, 79(8):2554–8.

[Hopfield and Tank, 1985] Hopfield, J. and Tank, D. (1985). Neural? computation of decisions in optimization problems. *Biological Cybernetics*, 52(3):141–152.

[Hopfield and Tank, 1986] Hopfield, J. and Tank, D. (1986). Computing with neural circuits: a model. *Science*, 233(4764):625–633.

[Hornik et al., 1989] Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366.

[Ito and Tani, 2004] Ito, M. and Tani, J. (2004). On-line Imitative Interaction with a Humanoid Robot Using a Dynamic Neural Network Model of a Mirror System. *Adaptive Behavior*, 12(2):93.

[Jaeger and Haas, 2004] Jaeger, H. and Haas, H. (2004). Harnessing Nonlinearity: Predicting Chaotic Systems and Saving Energy in Wireless Communication. *Science*, 304(5667):78–80.

[Jordan, 1986] Jordan, M. (1986). Attractor dynamics and parallelism in a connectionist sequential machine. *Proceedings of the Eight Annual Conference of the Cognitive Science Society*, pages 513–546.

[Kandel et al., 1991] Kandel, E., Schwartz, J., and Jessell, T. (1991). *Principles of Neural Science (third edition)*. Appelton and Lange, Norwalk, Conn. (USA).

[Kandel et al., 1995] Kandel, E. R., Schwartz, J. H., and Jessell, T. M. (1995). *Essentials of Neural Science and Behavior*. Appleton & Lange, Norwalk, Connecticut.

[Kleinfeld, 1986] Kleinfeld, D. (1986). Sequential State Generation by Model Neural Networks. *Proceedings of the National Academy of Sciences of the United States of America*, 83(24):9469–9473.

[Kohonen, 1982] Kohonen, T. (1982). Self-organized formation of topologically correct feature maps Biol. *Biological Cybernetics*, 43:59–69.

[Kohonen, 1988] Kohonen, T. (1988). The 'Neural' Phonetic Typewriter. *Computer*, 21(3):11–22.

[Kosko, 1992] Kosko, B. (1992). *Neural networks and fuzzy systems: a dynamical systems approach to machine intelligence*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA.

[Lambrinos, 1995] Lambrinos, D. (1995). Navigating with an Adaptive Light Compass. *Proc. ECAL (European Conference on Artificial Life)*.

[Le Cun et al., 1990] Le Cun, Y., Denker, J., and Solla, S. (1990). Optimal brain damage. *Advances in Neural Information Processing Systems*, 2:598–605.

[Minsky and Papert, 1969] Minsky, M. and Papert, S. (1969). *Perceptrons: an introduction to computational geometry*. MIT Press Cambridge, Mass.

[Pfeifer and Bongard, 2007] Pfeifer, R. and Bongard, J. (2007). *How the Body Shapes the Way We Think: A New View of Intelligence*. Cambridge, Mass. MIT Press.

[Pfeifer and Scheier, 1999] Pfeifer, R. and Scheier, C. (1999). *Understanding intelligence*. Cambridge, Mass. MIT Press.

[Pfister et al., 2000] Pfister, M., Behnke, S., and Rojas, R. (2000). Recognition of Handwritten ZIP Codes in a RealWorld Non-Standard-Letter Sorting System. *Applied Intelligence*, 12(1):95–115.

[Poggio and Girosi, 1990] Poggio, T. and Girosi, F. (1990). Regularization Algorithms for Learning That Are Equivalent to Multilayer Networks. *Science*, 247(4945):978–982.

[Pomerleau, 1993] Pomerleau, D. (1993). *Neural Network Perception for Mobile Robot Guidance*. Kluwer Academic Publishers Norwell, MA, USA.

[Port and Van Gelder, 1995] Port, R. and Van Gelder, T. (1995). *Mind as Motion: Explorations in the Dynamics of Cognition*. Cambridge, Mass. MIT Press.

[Reeke George et al., 1990] Reeke George, N., Finkel Leif, H., and Sporns Olaf, E. (1990). Synthetic Neural Modeling: A Multilevel Approach to the Analysis of Brain Complexity, Chap. 24. *Edelman Gerald M. Gall Einar W. Cowan Maxwell W.(eds.), Signals and sense-Local and Global Order in Perceptual Maps, New York, Wiley-Liss*.

[Ritter and Kohonen, 1989] Ritter, H. and Kohonen, T. (1989). Self-organizing semantic maps. *Biological Cybernetics*, 61(4):241–254.

[Ritter et al., 1991] Ritter, H., Martinetz, T., and Schulten, K. (1991). Neuronale Netze: Eine Einführung in die Neuroinformatik selbstorganisierender Netzwerke. *Bonn: Addison-Wesley*.

[Ritz and Gerstner, 1994] Ritz, R. and Gerstner, W. (1994). Associative binding and segregation in a network of spiking neurons In: Models of Neural Networks II: Temporal aspects of Coding and Information Processing in Biological Systems, Domany, E., van Hemmen, JL, Schulten, K.

[Rosenblatt, 1958] Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychol Rev*, 65(6):386–408.

[Rumelhart et al., 1986a] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986a). Learning internal representation by error propagation. *Parallel Distributed Processing*, 1.

[Rumelhart et al., 1986b] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986b). Learning representation by back-propagating errors. *Nature*, 323:533–536.

[Rumelhart et al., 1988] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1988). Learning representations by back-propagating errors. In *Neurocomputing: foundations of research*, pages 696–699. MIT Press, Cambridge, MA, USA. 0-262-01097-6.

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

[Strogatz, 1994] Strogatz, S. (1994). *Nonlinear Dynamics and Chaos: With Applications to Physics, Biology, Chemistry, and Engineering*. Perseus Books.

[Sutton and Barto, 1998] Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. The MIT Press.

[Utans and Moody, 1991] Utans, J. and Moody, J. (1991). Selecting neural network architectures via the prediction risk: application to corporate bond rating prediction. *Artificial Intelligence on Wall Street, 1991. Proceedings., First International Conference on*, pages 35–41.

[Vapnik and Červonenkis, 1979] Vapnik, V. and Červonenkis, A. (1979). *Theorie der Zeichenerkennung*. Berlin: Akademie-Verlag (original publication in Russian, 1974).

[Vapnik and Chervonenkis, 1971] Vapnik, V. N. and Chervonenkis, A. Y. (1971). On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, 16(2):264–280.

[Visell, 2009] Visell, Y. (2009). Tactile sensory substitution: Models for enaction in hci. *Interacting with Computers*, 21(1-2):38 – 53. Special issue: Enactive Interfaces.

[von Melchner et al., 2000] von Melchner, L., Pallas, S., and Sur, M. (2000). Visual behaviour mediated by retinal projections directed to the auditory pathway. *Nature*, 404(6780):871–6.

[Werbos, 1974a] Werbos, P. (1974a). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, Cambridge, MA.

[Werbos, 1974b] Werbos, P. (1974b). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Ph.D. Thesis, Harvard University.

[Wilson and Pawley, 1988] Wilson, G. and Pawley, G. (1988). On the stability of the Travelling Salesman Problem algorithm of Hopfield and Tank. *Biological Cybernetics*, 58(1):63–70.