Chapter Two

Fundamentals of Neural Network Models

This chapter contains a technical presentation of neural network models and of how they are rooted in neurophysiology. The aim is to provide the reader with a basic understanding of how neural network models work and how to build them. We begin with basic concepts of neurophysiology, which are then summarized in a simple mathematical model of a network node. We continue showing different ways to connect nodes into networks and discuss what different networks can do. Then we consider how to set network connections so that the network behaves as desired. We conclude with a section on computer simulation of neural networks.

2.1 NETWORK NODES

In Chapter 1 we saw that neural network models consist of interconnected nodes that are inspired by biological neurons. Nodes in a neural network model are typically much simpler than neurons, including only features that appear important for the collective operation of the network. A minimal set of such features may be summarized as follows:

- Nodes can be in different states, including different levels of activity and possibly different values of internal variables.
- Nodes are connected to each other, and through these connections, they influence each other.
- The influence of a node on others depends on the node's own activity and on properties of the connection (excitatory or inhibitory, weak or strong, etc.).

Below we provide a basic sketch of how neurons operate (referring to Dayan & Abbott 2001; Kandel et al. 2000; and Simmons & Young 1999 for further details) and then we introduce the simple node model that we will use throughout this book.

2.1.1 Neurons and synapses

Neurons can be of many different types and can interact in different ways (Figure 2.1; Gibson & Connors 2003; Kandel et al. 2000; Simmons & Young 1999; Toledo-Rodriguez et al. 2003). Neurons have output pathways by which they influence other neurons and input pathways through which they are influenced. We begin by reviewing the operation of spiking neurons and their interactions through chemical synapses. Then we consider nonspiking neurons and interactions through electrical synapses and neuromodulators.

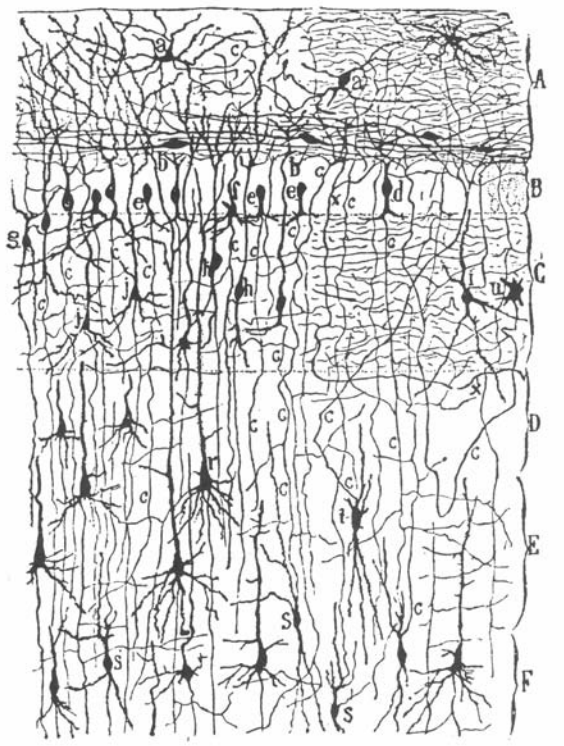

Figure 2.1 Part of the neural network of the inferior occipital cortex of a rabbit of eight days. Uppercase letters label anatomically distinguishable cortical layers, lowercase letters label different neuron types. Drawing by Ramón y Cajal, reprinted from De Felipe & Jones (1989).

The distinguishing feature of a *spiking neuron* is that it can generate electrical pulses called *spikes* or *action potentials*. These can travel along the neuron's axon, an elongated branching structure that grows out of the cell body. Axons can reach out for distances between a few millimeters and up to tens of centimeters, thus enabling both short- and long-distance interaction between neurons. The numbers of these interactions are often staggering. For instance, each cubic millimeter of mouse cortex contains about 10^5 neurons and an estimated 1 to 3.5 km of axons, whereby each neuron connects to about 8000 others (Braitenberg & Schüz 1998; Schütz 2003).

Neurons connect to each other by structures called *synapses*. At a synapse, the axon of one neuron (called *presynpatic*) meets a *dendrite* of another neuron (called *postsynaptic*). Dendrites form a structure similar to the axon but usually shorter and with more branches. While axons are output pathways, dendrites are input pathways. When a spike traveling along the axon of the presynaptic neuron reaches a chemical synapse, it causes the release of chemicals (*neurotransmitters*) that can excite or inhibit the postsynaptic neuron, i.e., favor or oppose the generation of a

spike by altering the electrical potential of the neuron. The magnitude of this effect depends on the amount of neurotransmitter released by the presynaptic neuron, as well as on the sensitivity of the postsynaptic neuron to the neurotransmitter. Altogether, these factors determine the "strength" of a synapse, i.e., the extent to which the presynaptic neuron is capable of influencing the postsynaptic one. Fundamental work has related basic learning phenomena such as habituation and classical conditioning to changes in the number and efficacy of synapses rather than to changes in patterns of connections between neurons (Chapter 4, Kandel et al. 2000).

The excitatory or inhibitory effect of the neurotransmitters released by the presynaptic neuron is transient. A simple picture of spiking neurons is that they keep track of excitatory and inhibitory inputs that arrive within a short time window, extending up to a few tens of milliseconds in the past. Whenever the balance of excitatory and inhibitory input within this time window overcomes a threshold, a spike is generated. Thus a neuron generates more spikes per unit time the larger the difference between excitatory and inhibitory input. Neuron activity is often expressed in terms of spike frequency, measured in Hertz (1 Hz = 1 spike per second). Physiology limits the spike frequency of most neurons to a maximum of a few hundred Hertz, although rates close to 1000 Hz can be observed in some neurons.

It is often of great importance to know how neuron activity changes when input varies. This holds also for receptor cells that convert light, chemical concentrations, sound waves, etc. into signals that are passed on to other neurons. In general, neurons respond nonlinearly to change in input; i.e., increasing the input in fixed steps does not increase output by the same amount for every step (Figure 2.2; Simmons & Young 1999; Torre et al. 1995; and Simmons & Young 1999 for reviews).

As mentioned earlier, *nonspiking neurons* also exist. Their influence on other neurons depends continuously on the neuron's electrical potential, e.g., through continuous changes in the rate of neurotransmitter release (Burrows 1979; Simmons & Young 1999). Likewise, chemical synapses are not the only way neurons interact. For instance, neurons may be joined by electrical synapses, or *gap junctions*. These allow neurons to exchange electric currents, thereby altering each other's electrical potential without the occurrence of spikes. Electrical synapses are often found in invertebrate nervous systems or in sense organs in vertebrates (Simmons & Young 1999). Recently, they have been discovered in the cortex of mammals (Gibson & Connors 2003). Neurons also influence each other by releasing relatively long-lived substances, called *neuromodulators*, into the extracellular fluid. These can change the properties of neurons over large regions of the nervous system, sometimes for hours. For instance, some neurons may become more excitable and others less so (Dickinson 2003; Harris-Warrick & Marder 1991; Hasselmo et al. 2003).

The activity of neurons depends on internal factors as well as external input. For instance, neurons respond to changes in input with a characteristic delay (Figure 2.3, left). Furthermore, the response to a constant input is often not constant at all (Wang & Rinzel 2003). A neuron may *habituate*, i.e., decrease its output to constant input (Figure 2.3, right), or it may produce oscillating activity. Neurons may also be spontaneously active in the absence of any input, as will be considered in Section 3.8 relative to the generation of motor patterns.

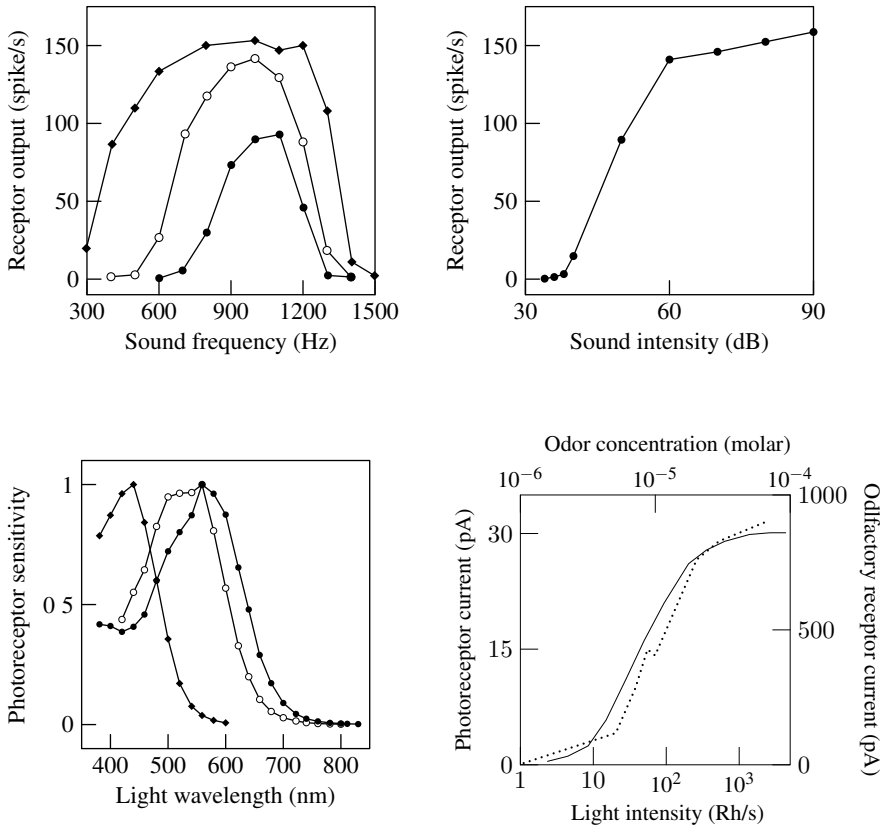

Figure 2.2 Responses of neurons to changes in physical characteristics of the input. Top left: Response of a ganglion cell in the ear of a squirrel monkey to sounds varying in frequency for three different intensities: 50 dB (closed circles), 60 dB (open circles) and 80 dB (diamonds). Top right: Response of the same cell to sounds of varying intensity and frequency of 1100 Hz (data from Rose et al. 1971). Bottom left: Relative sensitivity to lights of different wavelengths of the three kinds of cones in the retina of *Macaca fascicularis* (data from Baylor et al. 1987). Bottom right: Output of a newt (*Triturus cristatus*) photoreceptor to light varying in intensity (continuous line, left and bottom scales; data from Forti et al. 1989), and output of an olfactory receptor to different odorant concentrations (dotted line, scales at top and right; data from Menini et al. 1995). Note the similar shape of the two curves and of the curve in the panel above.

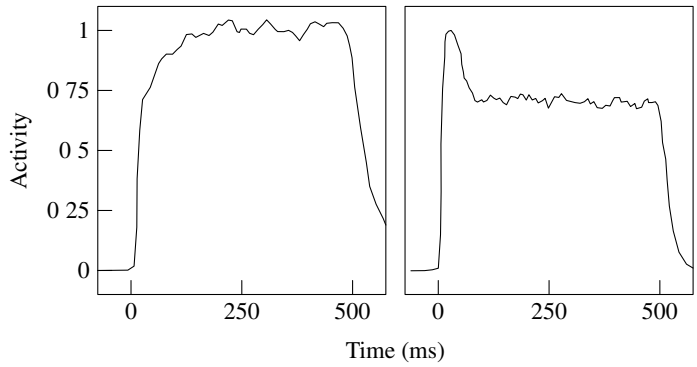

Figure 2.3 Responses of dark-adapted photoreceptors to a flash of light of 500 ms. Left: A cranefly (*Tipula*) photoreceptor approaches a steady-state activity rather slowly (activity is expressed as proportion of peak activity). Right: A fleshfly (*Sarcophaga*) photoreceptor reacts with a fast activity peak followed by lower steady-state activity (activity is expressed as proportion of maximum activity). Data from Laughlin (1981).

2.1.2 The basic node model

We now define mathematically network nodes and connections. Nodes in a network are indexed by integer numbers; a generic node is indicated by a lowercase letter such as i or j. The activity of node i is a (nonnegative) number written z_i. Its biological counterpart is the short-term average spiking frequency of a neuron or, for nonspiking neurons, the difference in electrical potential between the outside and inside of the neuron (see also Dayan & Abbott 2001; §1.2). The activities of all network nodes can be collected in an *activity vector* $\mathbf{z}$:

$$\mathbf{z} = \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_N \end{pmatrix} \tag{2.1}$$

where N is the number of nodes. Nodes are connected to each other via weights, continuous numbers that correspond to the strength of biological synapses. The weight that connects node j to node i is written W_{ij}. To indicate compactly all weights in the network, we introduce the *weight matrix* $\mathbf{W}$, which can be represented as an array with N rows and N columns:

$$\mathbf{W} = \begin{pmatrix} W_{11} & W_{12} & \cdots & W_{1N} \\ W_{21} & W_{22} & \cdots & W_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ W_{N1} & W_{N2} & \cdots & W_{NN} \end{pmatrix} \tag{2.2}$$

A first key assumption of neural network models is that node j conveys to node i an input equal to the product $W_{ij}z_j$. This models the biological fact that a presynaptic

neuron influences a postsynaptic one in a way that depends jointly on the presynaptic neuron's activity and on the strength of the synapse. The total input received by node i is written y_i and is calculated as the sum of all the inputs received:

$$y_i = \sum_j W_{ij} z_j \tag{2.3}$$

where the index j runs over all nodes in the network. Absence of a particular connection is expressed simply as $W_{ij} = 0$. Equation (2.3) is written more compactly as a relationship between vectors:

$$\mathbf{y} = \mathbf{W} \cdot \mathbf{z} \tag{2.4}$$

where $\mathbf{y}$ is the vector of all inputs, and $\mathbf{W} \cdot \mathbf{z}$ is the *matrix product* between $\mathbf{W}$ and $\mathbf{z}$. That is, $\mathbf{W} \cdot \mathbf{z}$ is simply a short way of writing the vector whose element i is $\sum_j W_{ij} z_j$ (the application of matrix and vector algebra to neural networks is detailed in Dayan & Abbott 2001 and Haykin 1999).

A second key assumption regards how input received by a node causes node activity. The function f that expresses the relationship between node input and node activity is called the *transfer function*:

$$z_i = f(y_i) \tag{2.5}$$

From a biological perspective, the transfer function embodies hypotheses about how neurons respond to input. In practice, however, the transfer function should also be easy to work with. The simplest transfer function is the *identity function*, by which node output equals node input:

$$f(y) = y \tag{2.6}$$

This function is used in mathematical analyses of neural network models because of its simplicity, but it has disadvantages if we recall that node activity should model the spiking rates or electrical potentials of neurons. Both these quantities are constrained within a range, and moreover, spiking rates cannot be negative. This results in curvilinear relationships between input magnitude and neuron activity (Figure 2.2) that are not captured by equation (2.6). It is also important that a network using only the identity transfer function can realize only a restricted set of input-output mappings (see Sections 2.2.1.1 and 2.3.1.1).

A simple solution is to use *logistic functions*. Some examples are shown in Figure 2.4. The output of a logistic function is always positive and less than 1; the latter can be interpreted as corresponding to the maximum activity of a neuron. Logistic functions have the form

$$f(y) = \frac{1}{1 + \mathrm{e}^{-a(y-b)}} \tag{2.7}$$

where the parameters a and b may be set to approximate empirical transfer functions. By tuning them, it is possible to regulate node activity to null input and how steeply activity rises as input increases (Figure 2.4, see also Figure 2.2). It is clear from Figure 2.2, however, that not even logistic functions satisfactorily describe all kinds of neurons and that especially to model receptors, different functions are needed. This issue will be taken up in Chapter 3.

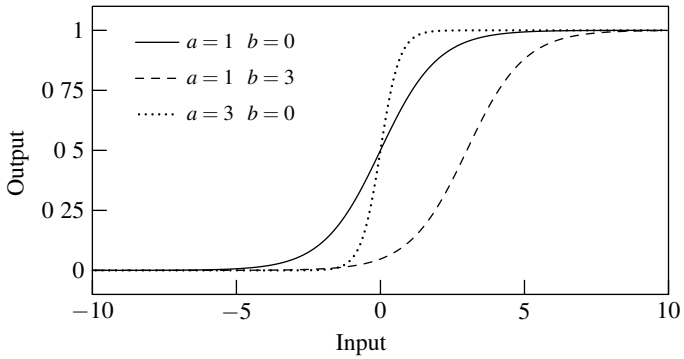

Figure 2.4 The logistic function, equation (2.7), for several choices of parameters. Note the similarity with the curves in the right part of Figure 2.2.

The node model introduced so far is summarized in Figure 2.5 and formally by

$$z_i = f\Big(\sum_j W_{ij} z_j\Big) \tag{2.8}$$

Accordingly, the operation of a whole network can be expressed symbolically as $\mathbf{z} = f(\mathbf{W} \cdot \mathbf{z})$. Obviously, this model is much simpler than neurons and synapses. Yet it has proven very useful to build models of behavior and nervous systems. In neural network studies, both simpler and more complex nodes are used, depending on the precise aim of the study. Simpler nodes sometimes are used to simplify mathematical analysis. An instance is a node having only two states, "active" and "inactive" (Amit 1989; McCulloch & Pitts 1943). More complex nodes have many uses, among which to investigate neuron physiology. For instance, Bush & Sejnowski (1991) reconstructed from photographs the branching structure of a few neurons, and Lytton & Sejnowski (1991) simulated them as complex electrical networks, accurately reproducing the observed spiking patterns. A network with such complex nodes, however, would be very difficult to build and to understand. Moreover, we are not guaranteed to get a better model of behavior simply by including more detail. We still ignore which features of neurons are crucial for nervous system operation and which are less important (Graham & Kado 2003). Thus we will start with the basic node described earlier, and add complexity only when needed to model behavioral phenomena.

2.1.3 More advanced nodes

This section shows how the basic node can be developed to include additional features of neurons. It can be skipped at first reading because most of our network models will use the basic node described earlier. We start by asking how to calculate node activity in response to an input $y(t)$ that varies with time. The simplest possibility is to construe time as a succession of discrete *time steps* and to compute node activity at time $t+1$ from node input at time t using equation (2.8):

$$z(t+1) = f(y(t)) \tag{2.9}$$

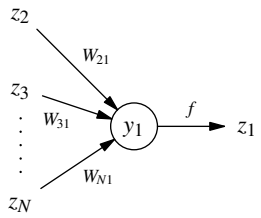

Figure 2.5 The basic node model. Node 1 receives input from nodes $2,\ldots,N$ through the weights $W_{12},\ldots,W_{1N}$. The total input received is $y_1 = \sum_{j=2}^{N} W_{1j} z_j$; it is passed through the transfer function f to yield node activity z_1.

The key feature of this equation is that node activity is determined solely by node input. The activity of neurons, on the other hand, may depend on factors such as the recent history of activity. In other words, neurons cannot instantly follow changes in input. For example, Figure 2.3 shows that a cranefly photoreceptor reacts with some delay to the onset of a light stimulus. This can be mimicked by letting $z(t+1)$ be a weighted average of $z(t)$ and $f(y(t))$:

$$z(t+1) = kf(y(t)) + (1-k)z(t) \tag{2.10}$$

Thus node activity in the next time step depends on current activity, as well as on current input, in a proportion given by the parameter k ($0 < k < 1$). If k is small, $z(t+1)$ tends to be close to $z(t)$; hence the node responds slowly to changes in input. Equation (2.10) is often written in terms of the *change* in activity from one time step to the next, $\Delta z(t) = z(t+1) - z(t)$. If we subtract $z(t)$ from both sides and write $1/k = \tau$, we get

$$\tau \Delta z(t) = f(y(t)) - z(t) \tag{2.11}$$

The parameter τ is called the *time constant* of the node: a smaller τ means that $z(t)$ can change more quickly. Figure 2.6 shows that the cranefly photoreceptor is well described by equation (2.11), with $\tau = 35$ ms. Equation (2.11) is also an approximation to simple models of node dynamics in continuous time (the main difference is that, in continuous time, the increment $\Delta z(t)$ is replaced by the time derivative $\mathrm{d}z(t)/\mathrm{d}t$).

How is equation (2.10) related to equation (2.9)? If the input is constant, the value $z(t)$ calculated from equation (2.11) eventually reaches a steady state equal to the one predicted by the latter (which is the same as equation 2.5). The steady state is obtained by setting $\Delta z(t) = 0$ (no change), in which case equation (2.11) becomes identical to equation (2.9). The steady state is approached at a rate given by τ; hence, if input changes on a time scale much longer than τ, the two equations are equivalent. This justifies using equation (2.9) when we are not interested in short-term responses.

We now show how further properties of neurons can be modeled by adding internal states. Our example is sensory habituation (or adaptation), shown in Figure 2.6. We introduce a single state variable h that represents the level of habituation, and assume that the node receives an inhibitory input of $-h$:

$$\tau \Delta z(t) = f\big(y(t) - h(t)\big) - z(t) \tag{2.12}$$

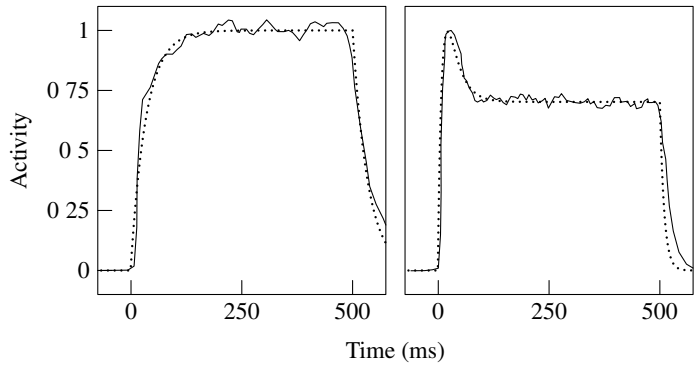

Figure 2.6 Two node models fitted to the data in Figure 2.3. Left: A cranefly (*Tipula*) photoreceptor is well described by equation (2.11), with $\tau = 35$ ms. Right: A fleshfly (*Sarcophaga*) photoreceptor is well described by equations (2.12) and (2.13), with $\tau = 10$ ms, T = 25 ms, $a = 0.5$.

Intuitively, h should increase when the node is strongly active and decrease when node activity is low. This is accomplished by the following equation:

$$\mathrm{T}\Delta h(t) = ay(t) - h(t) \tag{2.13}$$

where T is a second time constant, and a regulates the effect of habituation. The latter can be seen by calculating steady-state node activity arising from a constant input y. Setting $\Delta z(t) = 0$ and $\Delta h(t) = 0$ in the preceding equations, we obtain two equations for the steady-state values z and h:

$$\begin{aligned} z &= f(y-h) \\ h &= ay \end{aligned} \tag{2.14}$$

which imply $z = f((1-a)y)$. Hence the steady state of a habituating node is the same as that of a nonhabituating node whose input is reduced by a factor $1-a$. Figure 2.6 shows that this simple model can accurately describe the habituating photoreceptor of a fleshfly. Benda & Herz (2003) show that more complex kinds of habituation can be described with simple models. Models with a few internal variables have been developed also for other characteristics of neurons, such as spontaneous activity (Wang & Rinzel 2003).

2.2 NETWORK ARCHITECTURES

The pattern of connections among nodes defines the *network architecture* that is responsible for much of a network's properties. In the following we present a number of architectures relevant to behavioral modeling. We will first consider *feedforward networks*, so named because node activity flows unidirectionally from input nodes to output nodes, without any feedback connections. *Recurrent networks*, i.e., networks with feedback connections, will be considered later.

In describing neural networks, we retain notations used for behavior maps in general (Chapter 1). Input and output spaces can be defined by choosing some nodes as *input nodes* and others as *output nodes*. For instance, the former may model receptors and the latter motoneurons (connected to muscles) or neurons that secrete hormones or other chemicals (Chapter 3). Nodes that are neither input nodes nor output nodes traditionally are called *hidden nodes*. We write network input vectors as $\mathbf{x}_1$, $\mathbf{x}_2$, etc. Greek indices refer to different inputs and Latin indices to particular nodes. Thus $x_{\alpha i}$ is the activity of input node i elicited by the input $\mathbf{x}_\alpha$. Likewise, $r_{\alpha j}$ is the activity of output node j in response to input $\mathbf{x}_\alpha$. The letters y and z will be still used, as above, for the input and output of a generic node, respectively.

2.2.1 Feedforward networks

Figure 1.9 on page 19 shows some feedforward networks. Stimuli external to the network are assumed to activate the network's input nodes. This activation pattern is processed in a number of steps corresponding to node "layers." The activity in each layer is propagated to the next layer via a matrix of connections. Any number of layers may be present, although two- and three-layer networks are most common. Feedforward processing is an important organizational principle in nervous systems, e.g., in the way information from receptors reaches the brain. Although purely feedforward processing is rare, we shall see that these networks are helpful models of behavior. Moreover, feedforward networks are an important component of more complex architectures.

2.2.1.1 The perceptron

The first network architecture we consider is the so-called perceptron (Minsky & Papert 1969; Rosenblatt 1958, 1962). It consists of a single node receiving a number of inputs via a weight vector $\mathbf{W}$, as shown in Figure 1.9 on page 19. Owing to its simplicity, the perceptron is fully specified by a single equation:

$$r = f\Big(\sum_j W_j x_j\Big) = f(\mathbf{W}\cdot\mathbf{x}) \tag{2.15}$$

(see equation 2.8, and note that in this simple case we need only one index to label weights). The perceptron can be interpreted in several ways. It can model a very simple nervous system, where a number of receptors are connected directly to an output neuron, or it can be a building block for larger systems. In his seminal work, Rosenblatt considered the x_j's as the outputs of simple *feature detectors*, e.g., reporting about particular aspects of a visual scene (see the "analyzers" in Figure 1.11). The perceptron had to combine information about the presence or absence of individual features to reach yes-no decisions about the whole input pattern, e.g., about whether it belonged to some category. To this end, Rosenblatt used a transfer function assuming only the values 0 and 1 as follows:

$$f(y) = \begin{cases} 1 & \text{if } y > \theta \\ \\ 0 & \text{otherwise} \end{cases} \tag{2.16}$$

Table 2.1 A behavior map, interpreted as a simple model of feeding, that cannot be realized by a perceptron with two input nodes and one output node. The first two columns, each split in two, provide the mathematical description and interpretation of inputs and outputs. The third column shows how to choose perceptron parameters to realize single input-output mappings. However, there is no set of parameters that can realize all four mappings.

Input		**Desired output**		**Desired output obtained if:**
Values	**Interpretation**	**Value**	**Interpretation**	
(0,0)	All black	0	Don't eat	$\theta > 0$
(0,1)	Black and white	1	Eat	$W_2 > \theta$
(1,0)	White and black	1	Eat	$W_1 > \theta$
(1,1)	All white	0	Don't eat	$W_1 + W_2 < \theta$

where θ is a number serving as a *threshold*: if the total input $y = \mathbf{W} \cdot \mathbf{x}$ is above threshold, the output will be 1; otherwise, it will be 0. Rosenblatt also devised an algorithm to adjust the weights to correctly classify a set of input patterns into two categories. For instance, we could use this algorithm to solve the feeding control problem discussed on page 20.

Perceptrons, however, cannot realize all behavior maps. Consider a perceptron with two input nodes that model photoreceptors and an output node whose activity is interpreted as eating a prey. Each receptor reacts with 0 when no or little light is captured and with 1 when a lot of light is captured, e.g., when light reflected from a white object reaches the receptor. The model organism meets two kinds of potential prey. All-white prey is unpalatable and should be ignored. We assume that the presence of white prey reflects enough light to the retina to always activate both receptors. The second kind of prey instead should be eaten. Its presence activates only one receptor (e.g., because it is smaller), resulting in two possible input patterns: (0,1) and (1,0). The required behavior map is shown in Table 2.1 and is known as the *exclusive-or problem* (XOR). The last column in the table shows how the three perceptron parameters, W_1, W_2 and θ, must be chosen to realize each input-output mapping. For instance, the perceptron reacts to $(0,1)$ if $W_1 \times 0 + W_2 \times 1 = W_2$ is larger than θ (second table row). The problem is that the four conditions cannot all be satisfied by the same set of parameters: the sum $W_1 + W_2$ cannot be smaller than the positive number θ if each of W_1 and W_2 alone is bigger! The limits of perceptrons will be further discussed below (see also Minsky & Papert 1969).

2.2.1.2 From perceptrons to behavior

A few modifications make perceptrons more suitable for behavioral modeling, although their computational limitations are not removed. Rosenblatt's threshold mechanism (equation 2.16) implies that the same input always yields the same output. In contrast, an animal's reaction to a given stimulus typically is variable

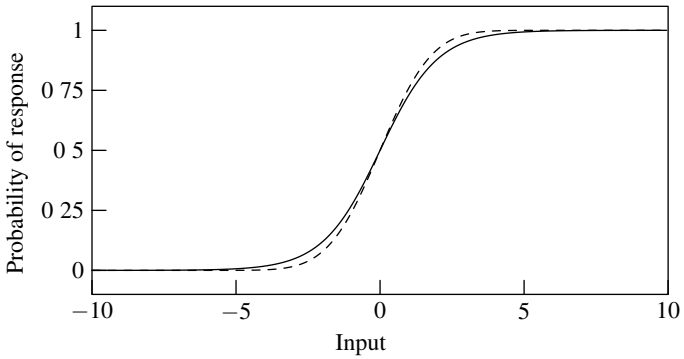

Figure 2.7 The probability that a node responds to an input for two different choices of the transfer function. Solid line: Equation (2.7), with $a = 1$ and $b = 0$. Dashed line: Equation (2.18), with $\sigma = 0.5$ and ε drawn from a standard normal distribution.

because the stimulus is not the only factor that influences behavior. We can remedy this shortcoming by following two slightly different routes. By the first route, we can use a transfer function f that rises smoothly from 0 to 1 (Figure 2.4) and interpret node activity as the probability that the animal responds to the stimulus:

$$\Pr(\text{reaction to } \mathbf{x}) = f(\mathbf{W} \cdot \mathbf{x}) \tag{2.17}$$

By the second route, the threshold mechanism is retained, but it is applied after adding to y a random number ε, e.g., drawn from a standard normal distribution. Formally, we replace equation (2.16) with

$$r = \begin{cases} 1 & \text{if } y + \sigma\varepsilon > \theta \\ 0 & \text{otherwise} \end{cases} \tag{2.18}$$

where σ is a positive number that regulates the importance of randomness. Output to $\mathbf{x}$ is now variable because ε is drawn anew each time $\mathbf{x}$ is presented. This, however, does not mean that behavior is wholly random because, according to equation (2.18), the probability of responding to $\mathbf{x}$ increases steadily as the input $y = \mathbf{W} \cdot \mathbf{x}$ increases. As shown in Figure 2.7, there can be very little difference in average behavior using either equation (2.17) or equation (2.18). In the former case, perceptron output is interpreted directly as the probability of responding. This is handy in theoretical analyses and to compare a model with data about average behavior. The latter transfer function can be used to generate individual responses to each input presentation, as needed in simulations of behavior (Section 2.3.4).

A very simple model of reaction to stimuli is obtained by considering equation (2.17) and equation (2.6) together:

$$\Pr(\text{reaction to } \mathbf{x}) = \mathbf{W} \cdot \mathbf{x} \tag{2.19}$$

with the further assumption that the probability is 0 if $\mathbf{W} \cdot \mathbf{x} < 0$ and 1 if $\mathbf{W} \cdot \mathbf{x} > 1$. In this way we can interpret the result of a straightforward computation as the probability that an animal reacts to a stimulus. We will see below that this simple model is very helpful to understand some important working principles of neural networks. In Chapter 3 we will apply equation (2.19) to many behavioral phenomena.

2.2.1.3 General feedforward networks

It is possible to build feedforward networks more powerful than perceptrons. For instance, we can add output nodes (Figure 1.9 on page 19). In studies of motor control, the output nodes may model motoneurons that control individual muscles. In this case, the sequence of output patterns generated by the network would correspond to sequences of muscle contractions (Section 3.8). When modeling behavior, it is often more convenient to interpret the activity of each output node as the tendency to perform a particular behavior pattern. One way to decide which behavior is actually performed is to assume that the most active node always takes control. Alternatively, we may interpret the activity of a node as the (relative) probability that the corresponding behavior be performed. We will see in Section 3.7 that both rules can emerge from reciprocal inhibition between the output nodes (possibly in the presence of random factors).

Feedforward networks can also be expanded with additional layers (Figure 1.9 on page 19). If $\mathbf{z}_i$ is the ith layer of nodes, and $\mathbf{W}^{(i)}$ is the weight matrix from layer i to layer $i+1$, the operation of a general feedforward network may be summarized by $\mathbf{z}_{i+1} = f(\mathbf{W}^{(i)} \cdot \mathbf{z}_i)$. Such networks, often called *multilayer perceptrons*, can solve a wider range of problems than networks with only an input and an output layer. In fact, multilayer perceptrons are able to realize arbitrary input-output maps given enough hidden units and provided the node transfer function is nonlinear (Haykin 1999). For instance, the behavior map in Table 2.1, which is impossible for perceptrons, can easily be realized by simple multilayer networks such as those shown in Figure 2.8. If the transfer function of hidden nodes is linear, it can be shown that a multilayer network is equivalent to one without hidden layers. The reason is that a single weight matrix is enough to perform any linear transformation between the input and output spaces. We will see in Section 3.4 that multilayer linear networks are nevertheless interesting to study. In Chapter 3 we will also use some nonlinear multilayer networks to model behavior that is too complex for linear networks.

2.2.2 Recurrent networks

Feedforward networks can realize any map with fixed input-output relationships but not maps in which the output to a given input depends on previous inputs. For instance, feedforward networks cannot distinguish the input sequences $\{\mathbf{x}_1, \mathbf{x}_2\}$ and $\{\mathbf{x}_2, \mathbf{x}_2\}$. The output to $\mathbf{x}_2$ at the second time step will be the same in both cases because the output of a feedforward network is determined solely by current input. Hence the time-processing abilities of feedforward networks are very limited. While some improvement is possible (Section 2.3.5), recurrent networks are a superior alternative to analyze input sequences and generate output sequences. The key architectural element in recurrent networks is *feedback loops*, or *recurrent connections*. This means that the activity of a node at a given time can be influenced by the activity of the same node at previous times. Usually this influence is mediated by the activity of other nodes at intervening times. In other words, while in feedforward networks only the weights serve as memory of the past, in recurrent networks

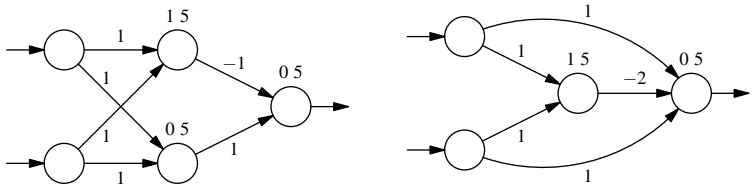

Figure 2.8 Two networks that realize the input-output map in Table 2.1 (exclusive-or problem, or XOR). Node transfer function is equation (2.16); numbers above each node are threshold values. The reader can check that among the input patterns (0,0), (1,1), (0,1) and (1,0), only the latter two activate the output node.

the activity of nodes is also a source of memory. Recurrency is a common feature of nervous systems and underlies their more sophisticated abilities. For instance, feedback is very common in the cerebral cortex, both within and between different cortical areas.

The behavior of recurrent networks can be very diverse (Amit 1989; Dayan & Abbott 2001; Hertz et al. 1991). In response to a constant input, a recurrent network may settle into a persistent pattern of activity (possibly a different one for different inputs), it may cycle repeatedly through a sequence of states, or it may exhibit irregular behavior. When the input varies in time, the network generally will not settle into one pattern of activity but will generate a sequence of output patterns, in principle of arbitrary complexity. Thus recurrent networks appear a promising tool to model time structures in animal behavior but also a tool that is not trivial to handle. In Section 2.3.5 we will show a few recurrent architectures and how to build a recurrent network with desired properties. In the following chapters we will apply recurrent networks to behavioral modeling.

We conclude with a technical note that can be skipped at first reading but should be considered when using recurrent networks. To work with recurrent networks, we need a way to calculate node activity as time proceeds, i.e., a dynamics for the network. The simplest possibility is to consider discrete time steps, and to use equation (2.8) to calculate the activity of all nodes at time $t+1$ based on activities at time t:

$$z_i(t+1) = f\Big(\sum_j W_{ij} z_j(t)\Big) \tag{2.20}$$

This simple dynamics is adequate in some cases but in general has drawbacks. Its main feature is that all nodes are updated simultaneously, as if a "global clock" synchronized the network (*synchronous dynamics*). By contrast, neurons in a nervous system can change their activity at any time, independently of what other neurons are doing. A change in one neuron then affects other neurons in the form of changed input (with some delay owing to spike travel times and synaptic transmission times). Given the same weights, synchronous dynamics can produce different results than more realistic continuous-time dynamics. Moreover, small changes to the update procedure or small amounts of noise can alter network behavior (Amit 1989). An obvious way to overcome these shortcomings is to use continuous time, but in many cases it is enough to modify equation (2.20) slightly. For instance, we

can use equation (2.11), with $y(t) = \sum_j W_{ij} z_j(t)$ (see Section 2.1.3 for details). The latter is in fact a discrete-time approximation of a simple continuous time model (since digital computers operate in time steps, such an approximation is always necessary in computer simulations). Another possibility is as follows. At each time step we choose a random ordering of the nodes. We then update the first chosen node. When we update the second node, we use the updated activity of the first node, and so on for all nodes. Since the random order is different at each time step, the procedure mimics the uncoordinated dynamics in real nervous systems and thus is called *asynchronous dynamics* (Amit 1989). Any result obtained in this way obviously is resistant to changes in the order of update operations.

2.3 ACHIEVING SPECIFIC INPUT-OUTPUT MAPPINGS

The input-output map realized by a network depends, of course, on the values of the weights. In practice, much of the work with neural networks is concerned with setting weights. Up to now we have considered problems in which suitable weights could be found either by intuition or by using simple mathematics. In this section we describe some powerful weight-setting methods that can deal with more complex problems. In the next section we describe procedures that organize network weights in response to incoming input rather than seeking to achieve a particular input-output map.

Weight-setting methods can be viewed either as models of real-world phenomena, i.e., how synapses change in nervous systems, or simply as a tool to build networks that handle a given task, without any additional claim. Here the focus is on the latter view, whereas Chapter 4 covers biological learning. Our goal is to build a network that produces a number μ of assigned input-output mappings:

$$m(\mathbf{x}_\alpha, \mathbf{W}) = \mathbf{d}_\alpha \qquad \forall \alpha = 1, \ldots, \mu \tag{2.21}$$

where m is the input-output map realized by the network, $\mathbf{W}$ denotes all network weights, and $\mathbf{d}_\alpha$ is the desired output pattern to input $\mathbf{x}_\alpha$. Equation (2.21) is a system of μ equations where the unknowns are network weights. The direct analysis of these equations is typically very challenging or plainly unfeasible. The only case that can be treated thoroughly with simple mathematics is linear networks, i.e., networks in which all nodes have a linear transfer function, such as equation (2.6). We discuss this case (with a final note on networks with nonlinear output nodes) before presenting more general weight setting techniques.

2.3.1 Setting weights in linear networks

If the transfer function of all nodes is linear, weights satisfying equation (2.21) can be found by formal methods, or it can be proved that they do not exist. We start with a linear perceptron, equation (2.19), and with a single input-output mapping. This corresponds to equation (2.21) with $\mu = 1$ and $m(\mathbf{x}_1, \mathbf{W}) = \mathbf{W} \cdot \mathbf{x}_1$. In this case it is easy to check that the single requirement $\mathbf{W} \cdot \mathbf{x}_1 = d_1$ is satisfied by a weight vector of the form

$$\mathbf{W} = c_1 \mathbf{x}_1 \tag{2.22}$$

with $c_1 = d_1/(\mathbf{x}_1 \cdot \mathbf{x}_1)$. Moreover, if we add to $\mathbf{W}$ a vector $\mathbf{u}$ such that $\mathbf{u} \cdot \mathbf{x}_1 = 0$, the result is not altered. In conclusion, the general solution to the task is of the form

$$\begin{cases} \mathbf{W} = \dfrac{d_1}{\mathbf{x}_1 \cdot \mathbf{x}_1}\mathbf{x}_1 + \mathbf{u} \\ \\ \mathbf{u} \cdot \mathbf{x}_1 = 0 \end{cases} \tag{2.23}$$

By extending this reasoning, it can be shown that the weight vector that realizes a number μ of input-output mappings can be written as

$$\begin{cases} \mathbf{W} = \displaystyle\sum_{\alpha=1}^{\mu} c_\alpha \mathbf{x}_\alpha + \mathbf{u} \\ \\ \mathbf{u} \cdot \mathbf{x}_\alpha = 0 \qquad \forall \alpha = 1, \ldots, \mu \end{cases} \tag{2.24}$$

The general expression for the c_α's, however, is not as simple as in equation (2.23). We show how to solve this problem in the case of two input-output mappings. This simple task will be used throughout this section to illustrate and compare different weight setting techniques. It may be helpful to think of a discrimination task where $\mathbf{x}_1$ has to be mapped to a high output and $\mathbf{x}_2$ to a low one, although the procedure is the same for all desired outputs.

Following equation (2.24), we guess that $\mathbf{W}$ should be of the form $c_1\mathbf{x}_1 + c_2\mathbf{x}_2$, so our task is to find the values of c_1 and c_2 that satisfy the requirements $\mathbf{W} \cdot \mathbf{x}_\alpha = d_\alpha$. If we substitute $c_1\mathbf{x}_1 + c_2\mathbf{x}_2$ for $\mathbf{W}$, the requirements read

$$\begin{aligned} (c_1\mathbf{x}_1 + c_2\mathbf{x}_2) \cdot \mathbf{x}_1 = c_1\mathbf{x}_1 \cdot \mathbf{x}_1 + c_2\mathbf{x}_2 \cdot \mathbf{x}_1 = d_1 \\ (c_1\mathbf{x}_1 + c_2\mathbf{x}_2) \cdot \mathbf{x}_2 = c_1\mathbf{x}_1 \cdot \mathbf{x}_2 + c_2\mathbf{x}_2 \cdot \mathbf{x}_2 = d_2 \end{aligned} \tag{2.25}$$

For brevity, we write $\mathbf{x}_\alpha \cdot \mathbf{x}_\beta = X_{\alpha\beta}$:

$$\begin{aligned} X_{11}c_1 + X_{12}c_2 = d_1 \\ X_{12}c_1 + X_{22}c_2 = d_2 \end{aligned} \tag{2.26}$$

(note that $X_{12} = X_{21}$). The preceding is a system of two linear equations in the unknowns c_1 and c_2. It can be solved by a variety of methods, yielding

$$\begin{aligned} c_1 &= \frac{d_1X_{22} - d_2X_{12}}{X_{11}X_{22} - X_{12}^2} \\ c_2 &= \frac{d_2X_{11} - d_1X_{12}}{X_{11}X_{22} - X_{12}^2} \end{aligned} \tag{2.27}$$

As an exercise, the reader can check that these expressions are really such that $\mathbf{W} \cdot \mathbf{x}_1 = d_1$ and $\mathbf{W} \cdot \mathbf{x}_2 = d_2$. If the network has to satisfy many input-output mappings, the same techniques apply, but the calculations get boring. Software for numeric or formal mathematics will come in handy, whereas textbooks on linear algebra will provide the theoretical foundation.

Note that equation (2.24) expresses the weight vector $\mathbf{W}$ in terms of the vectors $\mathbf{x}_\alpha$. This gives a concrete interpretation to the intuitive statement that the $\mathbf{x}_\alpha$'s have been "stored in memory." Moreover, it allows one to understand the output to a

generic input $\mathbf{x}$ in terms of how $\mathbf{x}$ relates to the $\mathbf{x}_\alpha$'s (and to the vector $\mathbf{u}$). This will be discussed in Chapter 3.

Equation (2.24) also holds if the output node is not linear. Such a network can be written as $r = f(\mathbf{W}\cdot\mathbf{x})$. We mention this case because a nonlinear output node is used often to limit network output between 0 and 1 and interpret it as probability of responding. The proof follows from noting that the requirement $d_\alpha = f(\mathbf{W}\cdot\mathbf{x}_\alpha)$ can be written $\bar{d}_\alpha = \mathbf{W}\cdot\mathbf{x}_\alpha$, where $\bar{d}_\alpha = f^{-1}(d_\alpha)$ and f^{-1} is the inverse function of f. Thus we are back to a linear problem, albeit with different d_α's. This means that different choices of f result in different values of the c_α's but do not alter the structure of the weight vector.

2.3.1.1 Limits and abilities of linear networks

We expand here our remarks about the perceptron's limitations (page 40) by asking when the task in equation (2.21) can be solved by a perceptron. We then extend the reasoning to general linear networks. It is easy to determine when the task *cannot* be solved. Note, in fact, that the definition of a linear network $\mathbf{r} = \mathbf{W}\cdot\mathbf{x}$ implies

$$\mathbf{r}(\mathbf{x}_\alpha + \mathbf{x}_\beta) = \mathbf{r}(\mathbf{x}_\alpha) + \mathbf{r}(\mathbf{x}_\beta) \tag{2.28}$$

because $\mathbf{W}\cdot(\mathbf{x}_\alpha + \mathbf{x}_\beta) = \mathbf{W}\cdot\mathbf{x}_\alpha + \mathbf{W}\cdot\mathbf{x}_\beta$ (from equation 2.8). In words, the output to a sum of inputs equals the sum of the outputs to each input (indeed, this is the technical meaning of *linear*). It is then clear that an input-output map where, say, $\mathbf{x}_\gamma = \mathbf{x}_\alpha + \mathbf{x}_\beta$ but $\mathbf{d}_\gamma \neq \mathbf{d}_\alpha + \mathbf{d}_\beta$ cannot be realized by a linear network. In general, if among the inputs $\mathbf{x}_1, \ldots, \mathbf{x}_\mu$ there is one, say, $\mathbf{x}_\beta$, that can be written as a weighted sum of the others, e.g.,

$$\mathbf{x}_\beta = \sum_{\alpha\neq\beta} b_\alpha \mathbf{x}_\alpha \tag{2.29}$$

(for some choice of the b_α's), then a weight vector $\mathbf{W}$ realizing the input-output map in equation (2.21) does not exist unless

$$\mathbf{d}_\beta = \sum_{\alpha\neq\beta} b_\alpha \mathbf{d}_\alpha \tag{2.30}$$

That is, the same relationship must hold among outputs that holds among inputs. Conversely, if it is impossible to write any of the $\mathbf{x}_\alpha$'s in the form of equation (2.29), the input-output map in equation (2.21) can be realized by a linear network for any choice of the $\mathbf{d}_\alpha$'s. Indeed, the reader may have noted that equation (2.27) is meaningless if $X_{11}X_{22} = X_{12}^2$ (fraction denominators vanish). This happens precisely when $\mathbf{x}_1 = b\mathbf{x}_2$, a special case of equation (2.29). In fact, in this case we have

$$\begin{aligned} X_{11} &= \mathbf{x}_1\cdot\mathbf{x}_1 \\ X_{22} &= b^2\mathbf{x}_1\cdot\mathbf{x}_1 \\ X_{12} &= b\mathbf{x}_1\cdot\mathbf{x}_1 \end{aligned} \tag{2.31}$$

so that $X_{11}X_{22} = X_{12}^2 = b^2(\mathbf{x}_1\cdot\mathbf{x}_1)^2$. Further information on this topic can be sought in linear algebra textbooks (sections on systems of linear equations).

2.3.2 Gradient descent methods (δ rule, back-propagation)

In this section and the next we introduce weight-setting methods that require a so-called objective function for their operation. These methods can work with non-linear networks of, in principle, arbitrary complexity. Indeed, the methods are very general and work with any input-output map with adjustable parameters (Mitchell 1996; Widrow & Stearns 1985). An *objective function* is simply a measure of network performance. It may indicate either how much the actual and desired maps differ (*error measure*) or how close they are (*performance measure*). It is a matter of convenience and partly of historical accident what measure is used in each particular case. If we use an error measure, the aim is to find weights that minimize it, whereas performance measures should be maximized. Setting weights in neural networks thus can be viewed as a function minimization or maximization problem (Aubin 1995; Widrow & Stearns 1985).

While many measures of performance are in use (Section 2.3.3), the *squared error* is by far the most popular error measure. Relative to the task (2.21), it is defined as follows. We introduce first the error relative to the input-output pair $(\mathbf{x}_\alpha, \mathbf{d}_\alpha)$ by summing the squared deviations between desired and actual output for each output node:

$$E_\alpha(\mathbf{W}) = \frac{1}{2}\sum_i (d_{\alpha i} - r_{\alpha i}(\mathbf{x}_\alpha))^2 \tag{2.32}$$

The notation $E_\alpha(\mathbf{W})$ emphasizes that the error is a function of the weights. The leading half serves merely to simplify notation later on. The square error relative to the whole task in equation (2.21) is defined as

$$E(\mathbf{W}) = \sum_\alpha E_\alpha(\mathbf{w}) \tag{2.33}$$

Clearly, the smaller the squared error, the closer we are to meeting the requirements of equation (2.21). Only when the squared error vanishes is the desired map established.

We are now ready to illustrate *gradient descent* methods of error minimization. Let us consider a network with only two weights, $\mathbf{W} = (W_1, W_2)$, and let us assume that the squared error $E(W_1, W_2)$ relative to the task is known for every value of W_1 and W_2. The triplet $(W_1, W_2, E(W_1, W_2))$ can be represented as a point in a three-dimensional coordinate space. When the weights vary, so does $E(W_1, W_2)$, and the point describes a surface known as the *error surface* (Figure 2.9). This allows us to visualize the values of the error for different weight choices. The quickest way to decrease the error is to travel on the error surface in the direction where the slope downward is steepest. To find what weight changes correspond to such a direction, one considers the *gradient* vector relative to the surface. This is written $\nabla E(\mathbf{W})$ and is defined as the vector whose components are the (partial) derivatives of $E(\mathbf{W})$ with respect to the weights:

$$\nabla E(\mathbf{W}) = \begin{pmatrix} \dfrac{\partial E(\mathbf{W})}{\partial W_{11}} \\ \vdots \\ \dfrac{\partial E(\mathbf{W})}{\partial W_{NN}} \end{pmatrix} \tag{2.34}$$

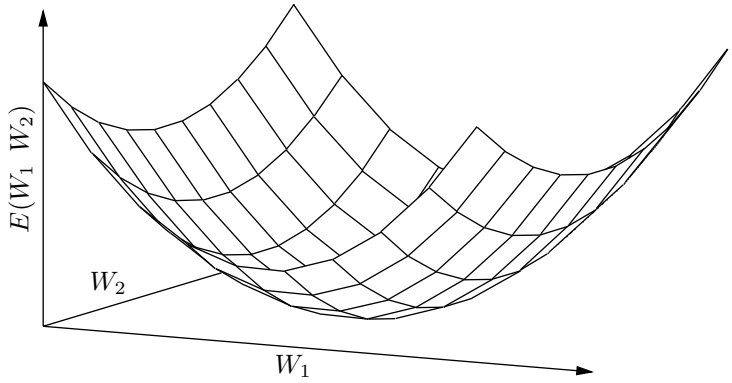

Figure 2.9 An error surface for a network with two weights. For a given task, the height of the surface at (W_1, W_2) is the error $E(W_1, W_2)$ relative to this particular choice of weights.

It can be shown that this vector points in the direction of steepest ascent. To descend the error surface, one thus should travel in the opposite direction. Additionally, the steps should not be too long: since the path to follow generally will be curved, the direction of travel needs to be updated regularly. This reasoning is formalized by writing the update ΔW_i to weight W_i as

$$\Delta W_i = -\eta \nabla_i E(\mathbf{W}) \tag{2.35}$$

where $\nabla_i E(\mathbf{W})$ is the ith component of the gradient vector (i.e., the derivative of $E(\mathbf{W})$ with respect to W_i), and η is a small positive number that calibrates step length. Once equation (2.35) has been applied to all weights, the gradient is calculated anew, then equation (2.35) is applied once more, and so on until error has decreased to a satisfactorily low figure. That is, gradient descent is an *iterative procedure*, and equation (2.35) is applied at each iteration. Note that although we started from an example with only two weights, equation (2.35) can be applied to networks with an any number of weights.

We must consider two important complications that are not apparent in Figure 2.9. In fact, the figure portrays a particularly favorable case where the error surface has a unique minimum, and the gradient always points in a direction that leads to the minimum along a rather short path. In general, neither is guaranteed to hold. The error surface may have many minima, of which only one or a few are satisfactory. Moreover, following the gradient may result in a very long path, which could be avoided with a fuller knowledge of the error surface. In defense of gradient methods, we can say that taking the path of steepest descent is a reasonable thing to do having only *local* information about the error surface (e.g., about how the error would change if the weights were changed slightly from their current values). The basic algorithm presented here can be improved in many ways, as explained, for instance, in Haykin (1999) and Golden (1996).

2.3.2.1 The δ rule

To compute the squared error and its gradient, one needs to know the output to all inputs and how it changes when changing any weight. A real input-output mechanism may not have all this information. For instance, input-output pairs are necessarily experienced in a sequence, and to remember previous inputs and outputs requires a memory external to the network. Without this memory, the best one can do is to compute the error relative to the current input, equation (2.32), rather than the full squared error, equation (2.33). This leads to the best known applications of gradient methods to neural networks (in particular feedforward networks). Applied to networks without hidden nodes, the method is known as the δ *rule*, thus named by McClelland and Rumelhart (1985) but known since Widrow and Hoff (1960) under the less appealing name of *least-mean-squares algorithm*. We begin with a single output node, in which case the instantaneous error is (we drop for simplicity the index α in equation 2.32):

$$E = \frac{1}{2}(d-r)^2 \tag{2.36}$$

where $r = \mathbf{W}\cdot\mathbf{x}$ is the response to $\mathbf{x}$. To calculate the derivative $\partial E/\partial W_i$, note first that the derivative $\partial r/\partial W_i$ is simply x_i. Hence

$$\begin{aligned}\frac{\partial E}{\partial W_i} &= -(d-r)\frac{\partial r}{\partial W_i}\\ &= -(d-r)x_i\end{aligned} \tag{2.37}$$

Thus, following equation (2.35), each weight is incremented by

$$\Delta W_i = \eta(d-r)x_i \tag{2.38}$$

which is the δ rule. It is so called because the difference $d-r$ was written δ by McClelland and Rumelhart (1985), yielding the even simpler expression

$$\Delta W_i = \eta\delta x_i \tag{2.39}$$

(having dropped the index α, we recall that $\mathbf{x}$ represents current input, i.e., one of the $\mathbf{x}_\alpha$'s for which a desired output d_α must be established; likewise, δ is the error relative to the current input).

The δ rule, it can be proved, will find a weight vector that, across the presentation of different inputs, keeps close to the one that minimizes the total squared error, provided the inputs are presented in random order and η is small enough (Widrow & Stearns 1985). Thus, in the case of networks without hidden layers, dealing with only one input at a time is not a serious restriction (because the error surface of these networks has a single minimum; see Widrow & Stearns 1985).

A reason why the δ rule is so popular is that equation (2.38) can be derived and understood intuitively without much mathematics as follows (Blough 1975; extending Rescorla & Wagner 1972). The term $d-r$ is the difference between desired and actual output, which is positive (negative) when actual responding is lower (higher) than desired responding and null when actual behavior matches desired behavior. That is, $d-r$ has the same sign as the desired change in response. Moreover, its absolute value is larger the more actual output departs from desired output. Thus, it

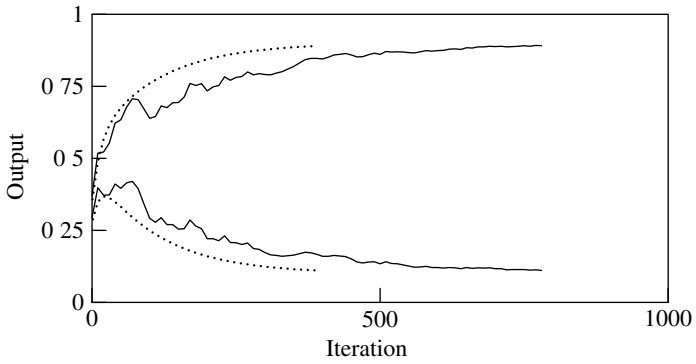

Figure 2.10 Comparison of δ rule learning (solid lines, equation 2.38) and full gradient descent (dotted lines, equation 2.40). A perceptron-like network with 10 input nodes and a linear output node is trained with either method to react with an output of 0.9 to an input $\mathbf{x}_1$ and with an output of 0.1 to $\mathbf{x}_2$.

can be used to command positive or negative weight changes of roughly appropriate magnitude. Multiplying $d - r$ by x_i has the further effect of changing more the weights that are linked to more active input nodes, under the intuitive justification that these weights contribute most to an incorrect output to $\mathbf{x}$.

The δ rule can easily solve the example task introduced earlier. The solid lines in Figure 2.10 are *learning curves* (network output as a function of algorithm iterations) from a simulation that uses the δ rule to solve the task. The dotted lines show that full gradient descent (based on the complete squared error, equation 2.33) solves the same task faster and more smoothly. The formula for full gradient descent is simply the δ rule formula summed over all input-output relationships:

$$\Delta W_i = \eta \sum_{\alpha} (d_\alpha - r_\alpha) x_{\alpha i} \tag{2.40}$$

The δ rule can also be extended to a nonlinear output node, where $r = f(\mathbf{W} \cdot \mathbf{x})$. Repeating the steps leading to equation (2.38), we get

$$\Delta W_i = \eta (d - r) f'(\mathbf{W} \cdot \mathbf{x}) x_i \tag{2.41}$$

The only addition is the term $f'(\mathbf{W} \cdot \mathbf{x})$. Actually, equation (2.38) can be used even when the output node is nonlinear (Hinton 1989). The δ rule can also be applied to networks with more output nodes. The weight matrix $\mathbf{W}$ is seen as made of weight vectors $\mathbf{W}_1, \ldots, \mathbf{W}_n$, where $\mathbf{W}_j$ connects the input nodes to output node j. Since these weight vectors are independent of each other, one simply applies the δ rule to each one, getting

$$\Delta W_{ij} = \eta (d_j - r_j) x_i \tag{2.42}$$

2.3.2.2 Back-propagation

Back-propagation is a generalization of the δ rule to feedforward networks with hidden layers. Although gradient descent is an old idea, it was not fully exploited

in the field of neural networks until publication of the back-propagation algorithm by Rumelhart et al. (1986). For simplicity, we begin with a network with a single hidden layer. We write **V**, the weights from input to hidden nodes; **h**, the activities of hidden nodes; and **W**, the weights from hidden to output nodes. The activity of output node k is thus

$$r_k = f\Big(\sum_k W_{ki} h_i\Big) \tag{2.43}$$

with

$$h_i = f\Big(\sum_j V_{ij} x_j\Big) \tag{2.44}$$

Technically, the back-propagation algorithm can be obtained by calculating the derivative of the preceding equations with respect to the weights, and then using the gradient descent formula, equation (2.35). The algorithm, however, can also be described starting from the δ rule. Note first that the hidden and output layers form a two-layer network that we can train by the δ rule:

$$\Delta W_{ki} = \eta f'(y_k) \delta_k h_i \tag{2.45}$$

where $y_k = \sum_k W_{ki} h_i$ is the input received by output node k, and $\delta_k = d_k - r_k$. What we lack is a formula to change the weights between input and hidden nodes. It seems that we cannot use the δ rule again because for hidden nodes *we have no concept of a desired output*, which means that we cannot calculate a δ term. The solution, with hindsight, is rather simple: the term δ_i, relative to hidden node i, is a weighted sum of the terms δ_k relative to the output nodes, where the weight of δ_k is exactly W_{ki}, i.e., the weight linking hidden node i to output node k (Haykin 1999; O'Reilly & Munakata 2000). Formally,

$$\delta_i = \sum_k \delta_k W_{ki} \tag{2.46}$$

Now that a δ term is defined, we can apply the δ rule as usual:

$$\Delta V_{ij} = \eta f'(y_i) \delta_i x_j \tag{2.47}$$

where $y_i = \sum_j V_{ij} x_j$. Equations (2.45) and (2.47) are the back-propagation algorithm. The algorithm can be generalized easily to networks with more than one hidden layer using δ terms of later layers to define δ terms in earlier layers. This backward construction of δ terms gave the algorithm its name.

2.3.3 Random search

Random search is an increasingly popular way of setting network weights, mostly because of its simplicity and power in many practical cases. Random search methods are also called *genetic algorithms* or *evolutionary algorithms* for reasons that will soon be clear (Holland 1975; Mitchell 1996). The basic procedure is as follows. From an initial network, typically with randomly set weights, some "mutants" are generated. Mutants differ from the initial network and from each other in a few weights, usually picked at random and modified by adding a small random

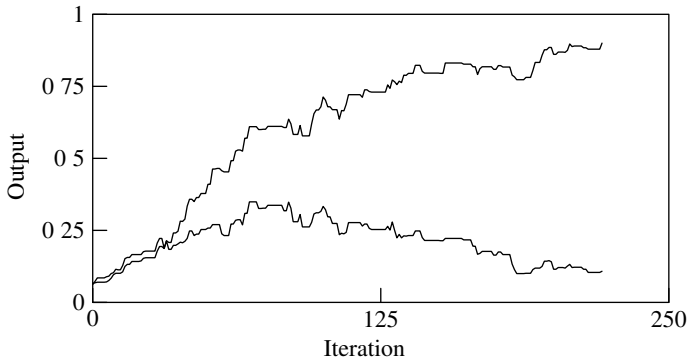

Figure 2.11 Random search simulation. A perceptron-like network with 10 input nodes and a linear output node is trained by random search to react with an output of 0.9 to an input $\mathbf{x}_1$ and with an output of 0.1 to $\mathbf{x}_2$.

number. The mutants and the initial network then are evaluated by means of a performance measure (often called *fitness*). The latter has been constructed so that it is maximized by desired behavior. The network yielding the highest performance is retained, and the process of mutation, evaluation and selection is repeated until a network capable of satisfactory performance is obtained or until one suspects that this will never happen. The basic algorithm can be modified in several respect, e.g., retaining more than one network from one iteration to the next or using different ways of generating mutants (which may include, e.g, changes in network architecture and node properties).

To illustrate random search, we consider our example task, using equation (2.15) with a logistic transfer function (equation 2.7). We need to define a performance function that reaches its maximum value when the task is solved. Since network output is limited between 0 and 1, the quantity $|d_\alpha - r_\alpha|$ (absolute value of the difference between the actual and the desired output for a given input-output mapping) can vary between 0 and 1, with 0 being its desired value. Hence the desired value of the quantity $1 - |d_\alpha - r_\alpha|$ is 1, and this quantity increases as r_α gets closer to d_α. If we need to establish two input-output mappings, we can define a performance function as follows:

$$p(\mathbf{W}) = \sqrt{(1 - |d_1 - \mathbf{r}(\mathbf{x}_1)|)(1 - |d_2 - \mathbf{r}(\mathbf{x}_2)|)} \tag{2.48}$$

(the square root lets $p(\mathbf{W})$ increases at a similar rate over its entire range). The maximum value of $p(\mathbf{W})$ is 1 and is reached only if $r_1 = d_1$ and $r_2 = d_2$. Equation (2.48) can be generalized easily to an arbitrary number μ of input-output mappings:

$$p(\mathbf{W}) = \sqrt[\mu]{\prod_{\alpha=1}^{\mu}(1 - |d_\alpha - \mathbf{r}(\mathbf{x}_\alpha)|)} \tag{2.49}$$

Figure 2.11 shows the results from a simulation using the simplest random search algorithm, whereby a single mutant network is generated at each iteration. Compared with gradient descent (Figure 2.10), progress on the task is more irregular.

Although a shortcoming in this simple case, in more complex tasks this may be a blessing. We have observed already that gradient descent methods may reach unsatisfactory local minima of the error surface. Random search is less prone (though not immune) to this kind of failure because occasional large mutations may lead out of local minima. Random search methods, moreover, can mimic genetic evolution, thereby allowing us to study the evolution of behavior (Chapter 5).

2.3.4 Learning from limited information (reinforcement learning)

The weight-setting methods described so far are based on detailed knowledge of the desired behavior map. Here we describe techniques that require less information. For instance, it may be enough to know whether the output is correct or not. These techniques are often called *reinforcement learning* because they resemble instrumental conditioning experiments in animal psychology (Barto 2003; Sutton & Barto 1998; see Chapter 4). In these experiments, animals do not (obviously) know what behavior the experimenter wants to train but can track what actions are rewarded or punished ("reinforced"). Reinforcement learning in artificial systems usually is achieved by increasing the probability of actions that, in a given situation, are followed by positive consequences and decreasing the probability of actions followed by negative consequences. In a self-contained learning device, this clearly requires a mechanism that identifies an event as positive or negative, as we will discuss in Chapter 4.

In our example task, reinforcement learning can be implemented as follows. First, network output must be interpreted as an action rather than as the probability of performing an action. This is achieved by using threshold transfer functions so that the output node is either active (interpreted as performing the action) or inactive. Moreover, we use a stochastic threshold node (equation 2.18) to generate variable output to the same input. This allows the network to try out different responses to the same stimulus, which is crucial to discover favorable responses. Let $\lambda = 1$ if reinforcement has been received and $\lambda = 0$ if not. Then the rule for changing the weights is

$$\Delta W_i = \begin{cases} \eta(r - \mathbf{W}\cdot\mathbf{x})x_i & \text{if } \lambda = 1 \\ \eta(1 - r - \mathbf{W}\cdot\mathbf{x})x_i & \text{if } \lambda = 0 \end{cases} \tag{2.50}$$

where $r = 1$ if the action has been performed, and $r = 0$ otherwise. To understand equation (2.50), note that both expressions have the same form of the δ rule, with r and $1 - r$ taken as desired responses (Widrow et al. 1973). Thus if performing the action has lead to reinforcement, it is taken as a goal. This works also to suppress responding to a input, in which case not performing the action is reinforced.

Now that network output is binary (react or not react), the d_i's cannot be interpreted as desired outputs. Their interpretation is rather that d_i is the probability that reinforcement will follow if the network responds to $\mathbf{x}_i$. Figure 2.12 shows the result of a simulation of equation (2.50). The outcome is that the network performs the action with probability d_i.

If the reinforcement can take on a continuous range of values, say, between 0 and 1, it is useful to combine the two parts of equation (2.50) into a single equation

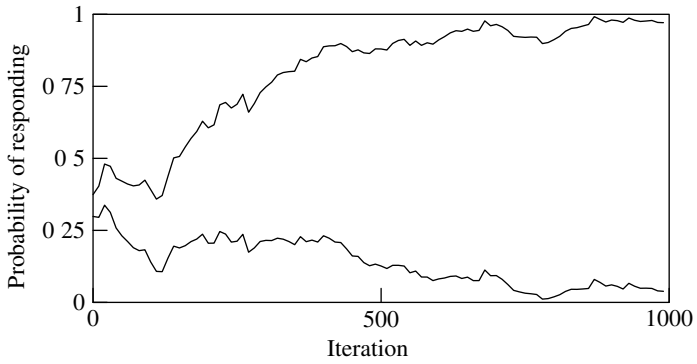

Figure 2.12 Reinforcement learning simulation relative to the same problem in Figures 2.10 and 2.11.

as follows (Barto 1995):

$$\Delta W_i = \eta\big(\lambda(r - \mathbf{W}\cdot\mathbf{x}) + \beta(1-\lambda)(1 - r - \mathbf{W}\cdot\mathbf{x})\big)x_i \tag{2.51}$$

If $\lambda = 1$ or $\lambda = 0$, this formula reduces to the upper or lower part of equation (2.50), respectively (but the learning rate for the lower part is $\eta\beta$ instead of η). If λ has an intermediate value, ΔW_i is a weighted sum of the two parts in equation (2.50), the weight being the parameter β. It can be shown that equation (2.51), called the *associative reward-penalty rule*, performs best for small values of β, e.g., 0.01 or 0.05. Nevertheless, the second part of equation (2.51) is important because it regulates weight changes when the action taken resulted in poor reinforcement.

2.3.5 Learning about time

So far we have discussed how to realize input-output maps with no time structure. Here we consider algorithms that can set weights so that desired output sequences are produced from given input sequences. We consider discrete time steps (Section 2.2.2), and we write $\mathbf{x}(t)$, the input at time t.

Different network architectures have been used to handle time. One idea is to maintain a feedforward architecture but to replicate the input layer so that it contains copies of the input at different times. This is called a *time-delay network*. Thus, if there are N input nodes receiving the input $\mathbf{x}(t)$, the network would have a second set of N nodes whose input is $\mathbf{x}(t-1)$, and so on (Figure 2.13, top). All techniques introduced to set weights in feedforward networks can be applied. For instance, a perceptron-like network may learn by the δ rule to react the sequence $\{\mathbf{x}_1, \mathbf{x}_2\}$ but not to $\{\mathbf{x}_2, \mathbf{x}_2\}$. This is impossible for a standard feedforward network (Section 2.2.2), but a time-delay network simply sees the two sequences as two different input patterns that thus can elicit different outputs. This simple approach to time, however, has a number of shortcomings. First, it may require a very large network. For instance, if we consider that signals in nervous systems propagate in millisecond times, we would need several hundred copies of the input layer to cover intervals of the order of 1 s. It does not appear that nervous systems manage time

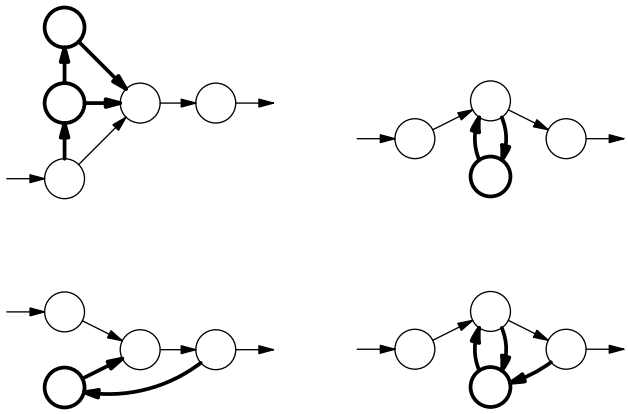

Figure 2.13 Four different modifications (thicker lines) to a feedforward network that add some time-processing abilities. Each circle represents a group of nodes. Top left: A network with no feedback but with multiple feedforward pathways that introduce a delay in the input. In this way the present activity of some nodes represents input received in the past (two delay steps are shown). Top right: The "simple recurrent network" of Elman (1990) that uses feedback between groups of hidden nodes to build internal states that depend on the time structure of the input. Bottom left: Network output is fed back as input. The network thus can combine information about current input and past output (Jordan 1986). Bottom right: A combination of the two previous architectures.

in this way (Buonomano & Karmarkar 2002). Second, everything is remembered with equal weight, even unimportant things.

Recurrent networks provide a superior alternative for temporal processing. These networks can build a more useful representation of input sequences than the simple bookkeeping of the past. Different recurrent architectures have been analyzed, e.g., networks with feedback between internal nodes (Elman 1990) or feedback from output nodes to internal nodes (Jordan 1986). Both features may be helpful, and both are important in nervous systems (Chapter 3). How can we establish a desired input-output map in a recurrent network? The question is difficult from the points of view of technique and biological realism alike. Gradient descent techniques may be applied that are conceptually like to the δ rule and back-propagation, but it is more difficult to calculate the error gradient (Doya 2003; Haykin 1999). The extension of random search techniques, on the other hand, is easier. We only need a performance function that considers *sequences* of inputs and outputs rather than single input-output mappings. Note, however, that the number of possible sequences grows exponentially with the number of time steps considered. Thus a naive extension of random search can be impractical. The very task of prescribing the correct output to every sequence of inputs can be prohibitive. Indeed, it is common that time-dependent tasks are not even phrased as an input-output map. For example, consider a network that must control a model arm so that it can reach toward objects. It would be practically impossible to analyze all possible input sequences

to determine what is the network action that is more likely to lead to reaching the object and then to teach this to the network. On the other hand, the task is clearly expressed as "reach the object." In such situations it is often most practical to define a performance function that considers the task at a higher level, e.g., a function that increases as the arm approaches the object (Nolfi & Floreano 2000). Temporal factors important for learning will be considered further in Chapter 4.

2.4 ORGANIZING NETWORKS WITHOUT SPECIFIC GUIDANCE

So far we have considered how to establish a number of given input-output mappings, but not all weight setting procedures are of this kind. Another possibility is to explore how weights develop when the network is exposed to sequences of inputs given a rule of weight change. For instance, experimental data or intuition about neurons and synapses may suggest a particular rule of weight change. We might then investigate whether the emerging weight organization resembles some feature of nervous systems. Alternatively, a rule of weight change may have been designed to produce a particular network organization, e.g., an input-output transformation that reduces noise in input patterns or enhances some of their features. In contrast with the preceding section, in these cases network organization emerges without specific feedback about whether a realistic or useful input-output map is being constructed. This has led to the term *self-organization*. It should be borne in mind, however, that whatever organization emerges depends crucially on the input fed to the network, in addition to properties of the network itself.

It is especially interesting to consider rules of weight change based on information that could be available at the junctions between neurons. These may include, for instance, state variables of both neurons and variables that describe the state of the extracellular environment, such as abundance of neuromodulators. These remarks will be expanded in Section 4.2. We now show a few examples of how interesting organizations of the weights can emerge from the following simple rule:

$$\Delta W_{ij} = \eta (z_i - a)(z_j - b) \tag{2.52}$$

where z_i and z_j are the activities of the nodes connected by W_{ij}, whereas a and b are two constants. Equation (2.52) implies that W_{ij} will increase if $z_i > a$ and $z_j > b$ or if $z_i < a$ and $z_j < b$ and decrease otherwise. We consider the network in Figure 2.14, with one output node that receives input from nodes arranged as a simple model retina. We feed this network with input patterns consisting of diffuse blobs of excitation (Figure 2.15, left) and apply equation (2.52) after each presentation. After repeating these steps a few hundreds times, we visualize the weights as a gray-scale image. Figure 2.15 shows three examples of how an initially random weight matrix can develop different organizations. The latter are often called *receptive fields* because they describe from which part of the retina a node receives input. Thus nodes with the receptive fields pictured in Figure 2.15 would react preferentially to images falling, respectively, on a well-delimited portion of the retina, on the left part of the retina, and anywhere on the retina but a specific portion. In a network with many such nodes, each one could react to stimuli on a different part

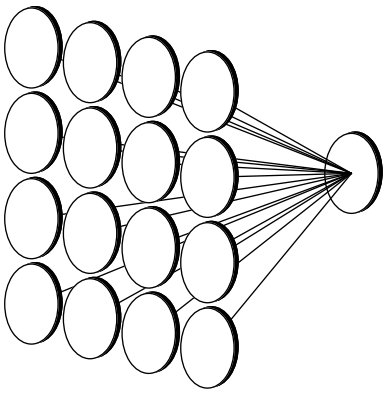

Figure 2.14 Network architecture used for simulations of self-organization. Input nodes are arranged in a two-dimensional grid simulating the physical arrangement of receptors on a retina.

of the retina, and further processing stages could use this information in tasks such as locating objects or orienting in the environment.

How can we produce a network with receptive fields spanning the whole retina? Simply adding output nodes that operate independently of each other is not enough. In fact, Figure 2.16 (left) shows that most nodes develop very similar receptive fields (this means that the sequence of inputs is a strong determinant of weight organization). We obtain a better result if, after presenting an input but before updating the weights, we set to zero the activity of all nodes except the most active one. Biologically, this may correspond to strong mutual inhibition between neurons. Receptive fields obtained in this way are spread more evenly across the retina, as shown in Figure 2.16 (right). To understand why, consider just two nodes and suppose that because of differences in initial weights, node 1 reacts more strongly to input $\mathbf{x}$ than node 2. Then, according to equation (2.52), the sensitivity of node 1 to $\mathbf{x}$ and similar inputs will be increased, and the sensitivity of node 2 (whose activity has been set to zero) to the same inputs will be decreased. Such weight changes make it even more likely that node 1 will be more active than node 2 to inputs similar to $\mathbf{x}$. In summary, we have a self-reinforcing process (positive feedback) that confers on the two nodes sensitivity to different input patterns.

2.5 WORKING WITH YOUR OWN MODELS

Students of animal behavior who wish to work with neural networks can choose among a diversity of software tools, many of which are surveyed by Hayes et al. (2003) and at http://www.faqs.org/faqs/ai-faq/neural-nets/part5. The focus of such software, however, is often not animal behavior. Here we offer a few guidelines for beginners. We broadly divide software in neural network simulators, general-purpose programming languages and software for scientific computation.

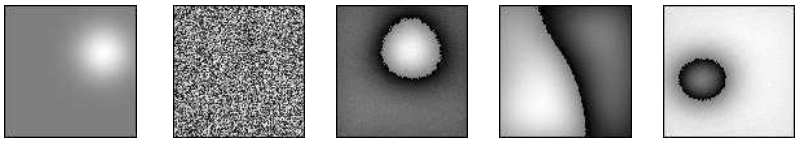

Figure 2.15 Simulations using equation (2.52) with the network in Figure 2.14 (input grid of 100×100 nodes). From left to right: A sample stimulus exciting nodes in a part of the input grid; initial weight configuration; three organizations of the weights, corresponding to three choices of the parameter b in equation (2.52), and $\eta = 5$, $a = 0.05$. From left to right: Excitatory center and inhibitory surround, excitatory and inhibitory hemifields, inhibitory center and excitatory surround. Simulations consisted of 1000 presentations of stimuli like the one shown, positioned randomly on the model retina. After each presentation, equation (2.52) was applied to all weights. A lighter shade of gray represents a stronger weight; negative weights are darker than the darkest gray in the sample stimulus.

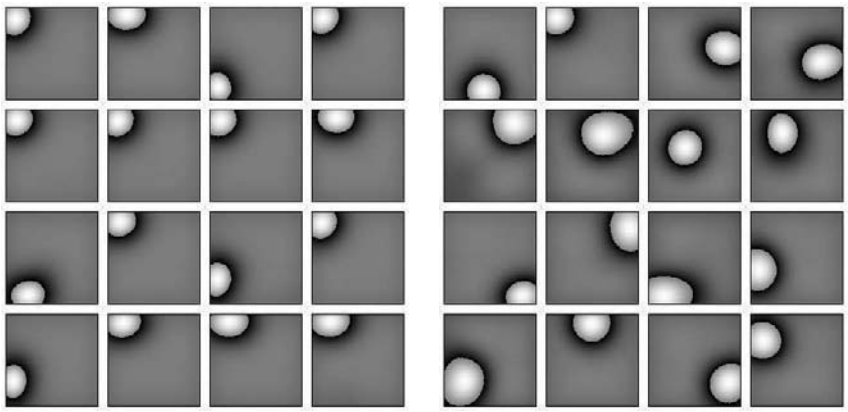

Figure 2.16 Weight organizations induced by equation (2.52) (with $a = 0.5$ and $b = 0.05$) in two different networks. Left: Receptive fields developing in a network like the one in Figure 2.14 but with 16 independent output nodes (all weights are updated independently at each input presentation). Right: Receptive fields developing in a similar network, but with strong inhibition between the 16 output nodes (only the weights feeding to the most active node are updated). In the latter case, receptive fields are spread more evenly.

```
#include <stdio.h> /* these two lines are needed to use the */
#include <math.h>  /* functions printf() and exp(), see below*/

/* this function calculates node input, see equation (2.3) */
/* W = weight vector, x = input vector, N = size of W and x */
float weighted_sum( float *W, float *x, int N ) {
  int i;
  float y = 0.;
  for( i=0; i<N; i++ ) y += W[i]*x[i];
  return y;
}

/* logistic node transfer function, see equation (2.7) */
float transfer_function( float y ) {
  return 1/(1+exp(-y));
}

/* a C program starts executing from the "main" function */
int main( void ) {
  float x[] = { 1, 1, 1, 1 };     /* input vector */
  float W[] = { 1, 0, 1, 0 };     /* weight vector */
  float y;                        /* input to output node */
  float z;                        /* activity of output node */
  y = weighted_sum( W, x, 4 );    /* calculate y */
  z = transfer_function( y );     /* apply the transfer function */
  printf( "Output is %f\n", z ); /* print result */
  return 0;                       /* end of the program */
}
```

Listing 2.1 A C program that simulates a two-layer network with $N = 4$ input nodes and 1 output node. Numbers are represented as variables of type `float` and vectors as arrays of such variables. Variables of type `int` represent integers (e.g., vector indices). The expression `x[i]` is the *i*th element of array `x` (C indices start from 0). The functions `weighted_sum` and `transfer_function` translate equation (2.3) and equation (2.7), respectively, and provide the basic structure to simulate two-layer networks. Explanatory comments are enclosed between `/*` and `*/`.

Neural network simulators are ready-to-use programs, often with graphical user interfaces, explicitly developed to study neural networks. Some of the most developed simulators are intended for neurophysiologically realistic simulations (Hayes et al. 2003 and references therein). They tend to consider many details of neurons, resulting in node models with many state variables and parameters that are, presently, of uncertain value to behavioral modeling (see the end of Section 2.1.2). Neural network simulators have also been developed within artificial intelligence, computer science and engineering. The drawback of such software is that it often requires some technical background and tends to focus on different questions than animal behavior. Two fairly general neural network simulators that, while requiring some study, are not too difficult for beginners and do not require programming are PDP++ (http://psych.colorado.edu/~oreilly/PDP++/PDP++.html) and JavaNNS (http://www-ra.informatik.uni-tuebingen.de/software/JavaNNS/welcome_e.html). The former focuses on (human) cognition, including behavior-level phenomena (O'Reilly & Munakata 2000), whereas the latter is more neutral regarding the interpretation of the network models.

It is also possible to do research on neural networks without specific software

using general-purpose programming languages such as Pascal, C, C++, etc. These have no knowledge of neural networks as such, yet simple neural network simulations can be implemented easily. For instance, to simulate a two-layer network with one output node, we need simply two arrays of numbers to store the input and weight vectors, one number for the response, as well as the ability to compute weighted sums and define transfer functions. As an example, Listing 2.1 shows a complete C program that calculates the output of a two-layer network with a single output node and weights $\mathbf{W} = (1,0,1,0)$ to the input $\mathbf{x} = (1,1,1,1)$. The program would not look too different in most other programming languages. People have also developed additions to the languages, called *libraries*, that allow programming of more complex simulations with little effort. Typically, these libraries provide data structures to represent various network architectures, as well as the ability to perform common tasks such as calculating network responses or gradient descent learning. Exploring such language extensions probably would be rewarding for readers with some programming experience or those who wish to start programming. Many neural network libraries are listed at http://www.faqs.org/faqs/ai-faq/neural-nets/part5.

Software for scientific computation is well suited for beginners wishing to simulate simple or moderately complex networks using a rather straightforward programming language. The key insight is that the operation of neural networks can be expressed in terms of vector and matrix algebra, for which extensive support exists. In the following we present a few examples in a syntax common to the programs Octave (http://www.octave.org, available free of charge) and Matlab (http://www.mathworks.com). Similar software, although with different programming syntax, are Scilab (http://www.scilab.org, available free of charge) and Mathematica (http://www.wolfram.com). Many of the models in later chapters require only a little more programming than the following examples, and we encourage the reader to try to reproduce some of our results. We do not expect readers without programming experience to understand each line of the code that follows but rather to get a flavor of neural network coding with scientific software.

2.5.1 Two-layer feedforward networks

Simple neural networks are coded rather compactly in scientific software. For instance, the following code is equivalent to Listing 2.1:

```
function r = f( y )
  r = 1 ./ (1+exp(-y)); % cf.equation (2.7)
end

x = [ 0 1 0 1]';        % input vector
W = [ 1 1 1 1];         % weight vector
r = f( W*x )            % cf. equation (2.17)
```

Note how the last line translates equation (2.17) almost literally, with `W*x` as equivalent to the notation $\mathbf{W} \cdot \mathbf{x}$ for a weighted sum. By convention, the result of a computation is printed out if the corresponding line does *not* end with a semicolon; hence the code above will display the value of `r`, i.e., `r = 0.88080`.

```
1  N = 10;          % number of input nodes
2  d = 0.9;         % desired output
3  t = 0.0001;      % target error
4  eta = 0.01;      % rate of weight change
5
6  x = rand(N,1);   % random input vector (uniform distribution)
7  W = randn(1,N); % random weight vector (normal distribution)
8
9  e = 2*t;                  % does not matter so long as e>t
10 while( e>t )
11   r = W*x;                % calculate output
12   W = W + eta*(d-r) * x'; % change weights
13   e = .5*(d-r)^2;         % calculate error
14 end
```

Listing 2.2 Code to establish a single input-output mapping by the δ rule in a two-layer network with $N = 10$ input nodes and 1 output node.

A technical comment: the prime (`'`) following the input vector `x` denotes a *column* vector (a matrix with one column and many rows), whereas `W` is a *row* vector (a matrix with one row and many columns). The distinction, while not crucial in this book, is important in matrix algebra and thus enforced in scientific software. In short, the matrix multiplication `W*x` is the weighted sum $\sum_i W_i x_i$ only if `W` is a row vector and `x` a column vector. Further explanations can be found in textbooks about linear algebra (Hsiung & Mao 1998).

It is easy to code networks with many output nodes. In fact, the transfer function defined earlier will return a vector of transformed values if we supply a vector of input values (this requires writing `./` rather than `/` on line 2 to perform the correct vector operation). In practice, this means that a network with more output nodes is coded simply by replacing the weight vector above with an appropriate weight matrix. A network with two output nodes, for instance, is coded like this:

```
1  x = [ 0 1 0 1]';
2  W = [ 1 1 1 1    % weights of 1st output node
3        2 2 2 2 ]; % weights of 2nd output node
4  r = f( W*x )
```

This will display `r = 0.88080 0.98201`, showing the output of both nodes. The extension to multilayer networks is also trivial. The following example calculates the output of a three-layer network with four input nodes, three hidden nodes and two output nodes:

```
1  x = [ 0 1 0 1]'; % input vector
2  W = [ 1 1 1 1    % weights from input to hidden nodes
3        2 2 2 2    % (3x4 matrix)
4        3 3 3 3 ];
5  V = [ 1 1 1      % weights from hidden to output nodes
6        2 2 2 ];   % (2x3 matrix)
7  r = f( V*f( W*x ) )
```

2.5.2 The δ rule

An algorithm to establish a number of input-output mappings by the δ rule is as follows:

```
1  N = 10;              % number of input nodes
2  M = 4;               % number of input-output mappings
3  t = 0.0001;          % target error
4  eta = 0.01;          % rate of weight change
5
6  rand("seed",time()); % initialize random number generator
7  d = rand(1,M);       % M random outputs
8  x = rand(N,M);       % M random inputs arranged in a matrix
9  W = randn(1,N);      % random weight vector
10
11 e = 2*t;
12 while( e>t )                    % while error is too big
13   e = 0;                        % reset error
14   for i = randperm(M)           % loop through inputs in random order
15     r = W * x(:,i);             % calculate output to x(:,i)
16     W = W + eta*(d(i)-r)*x(:,i); % change weights
17     e = e + .5*(d(i)-r)^2;      % add to error
18     end
19 end
```

Listing 2.3 Generalization of the code in Listing 2.2 to many input-output mappings. The expression `randperm(M)` on line 14 gives a random arrangement (permutation) of numbers from 1 to M. The expression `x(:,i)` on lines 15 and 16 is the ith column of the matrix `x`, i.e., the ith input vector.

1. Create input-output pairs.
2. Set weights at small random numbers.
3. Set total error to zero.
4. Arrange inputs in random order.
5. Loop through all inputs:
   a. Calculate output.
   b. Change the weights using the δ rule (Section 2.3.2.1).
   c. Calculate the error for this input, add to total error.
6. If the total error is low enough, end; else go to step 3.

Listing 2.2 is an implementation of this algorithm for a single input-output mapping, whereas Listing 2.3 is a generalization to an arbitrary number of input-output mappings (4 in the example). Lastly, Listing 2.4 shows how to implement full gradient descent by computing the full gradient in an auxiliary vector `G`, and then using equation (2.35) to update the weights.

2.5.3 Random search

Here is a description of the simplest random search algorithm:

1. Define a performance function.
2. Create two weight vectors. $\mathbf{W}_1$ represents the best vector found so far; $\mathbf{W}_2$ is an attempt to improve.
3. $\mathbf{W}_1$ is initialized with small random numbers.
4. $\mathbf{W}_2$ is made equal to $\mathbf{W}_1$.

```
M = 4;      % number of input-output mappings
N = 10      % number of input nodes
t = 0.0001; % target error
eta = 0.01; % rate of weight change

d = rand(1,M);       % M random outputs
x = rand(N,M);       % N random inputs arranged in a matrix
W = randn(1,10);     % random weight vector

e = 2*t;
while( e>t )                % while error is too big
  e = 0;                    % reset error
  G = zeros(size(W));       % gradient vector, same size as W
  for i = randperm(M)       % loop through inputs in random order
    r = W * x(:,i);         % calculate output to input i
    G = G - (d(i)-r)*x(:,i); % update gradient
    e = e + .5*(d(i)-r)^2;  % update error
  end
  W = W - eta*G;            % change weights, equation (2.40)
end
```

Listing 2.4 Modification of Listing 2.3 that uses full gradient descent instead of the δ rule (see Section 2.3.2.1). The auxiliary vector `G` stores the error gradient.

```
function p = perf( W, x, d ) % performance function: reaches
  p = 1 - abs( d - W*x );    % a maximum of 1 when W*x=d
end;

N = 10;         % number of input nodes
d = 0.9;        % desired output to x
t = .9999;      % target performance

x = rand(N,1);  % create a random input vector
W = randn(1,N); % create a random weight vector

p1 = perf( W1, x, d );  % initial performance

while ( p1<t )                % while performance too low
  W2 = W1;                    % let W2 equal W1
  i = ceil( N*rand() );       % choose a weight to mutate
  W2(i) = W2(i) + .01*randn(); % add small random number

  p2 = perf( W2, x, d ); % calculate performance of W2
  if( p2>p1 )            % accept W2 if it performs better
    W1 = W2;
    p1 = p2;
  end
end
```

Listing 2.5 Code for the simplest random search algorithm, establishing a single input-output mapping in a two-layer network. The expression `i=ceil(N*rand())` selects a random integer between 1 and N by selecting a random number in $[0,1]$, multiplying by N and finally rounding up to the next integer.

5. Add a small random number (equally likely to be positive or negative) to a randomly selected weight of $\mathbf{W}_2$.
6. Evaluate the performance of $\mathbf{W}_2$, and compare it with the performance of $\mathbf{W}_1$. If $\mathbf{W}_2$ performs better, it replaces $\mathbf{W}_1$ as the best vector so far.
7. If performance is satisfactory, end; otherwise return to step 4.

Listing 2.5 implements this algorithm for one input-output mapping.

CHAPTER SUMMARY

- Neural networks are mathematical models made of interconnected nodes that attempt to capture the most important features of neurons and synapses.
- Even simple networks can solve tasks that appear relevant for behavioral modeling.
- General feedforward networks can realize arbitrary input-output maps, provided the output to each input is fixed. A number of powerful weight-setting techniques exist to actually find the appropriate weights.
- Other weight setting techniques can be used to organize the network in response to incoming input (e.g., to encode statistical regularities) rather than to achieve a desired input-output mapping.
- Recurrent networks add the ability to process complex input sequences and generate time-structured output sequences. Recurrent networks often are more difficult to train, even though many techniques exist.
- If mathematical analysis of a neural network model is too difficult, we can simulate it on a computer using either ready-made software or programming our own.

FURTHER READING

Arbib MA, 2003. *The Handbook of Brain Theory and Neural Networks*. Cambridge, MA: MIT Press, 2 edition. A collection of about 300 articles on neural networks from different perspectives.

Dayan P, Abbott LF, 2001. *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. Cambridge, MA: MIT Press. Many topics treated formally but accessibly, from single neurons to large-scale network models.

Haykin S, 1999. *Neural Networks: A Comprehensive Foundation*. New York: Macmillan, 2 edition. A thorough introduction to almost anything technically important about neural networks, with good cross-field connections.

Mitchell M, 1996. *An Introduction to Genetic Algorithms*. Cambridge, MA: MIT Press. A general introduction to random search methods.