FIVE

Simple Perceptrons

In all the problems we have studied so far, it has been possible with a modicum of cleverness to figure out *a priori* how to choose appropriate connection strengths for a network. This may not always be practical however, because it may involve a large optimization problem or a large matrix inversion. It is often easier to adopt an iterative approach, in which appropriate w_{ij}'s are found by successive improvement from an arbitrary starting point. We can then say that the network is **learning** the task.

This chapter and the next two are devoted to **supervised learning**. Recall from Chapter 1 that there are two general learning paradigms, supervised and unsupervised learning. In supervised learning the network has its output compared with known correct answers, and receives feedback about any errors. This is sometimes called **learning with a teacher**; the teacher tells the network what the right answers are, or at least (in reinforcement learning) whether or not its own answers are correct. In unsupervised learning there is no teacher and no right and wrong answers; the network must discover for itself interesting categories or features in the input data. Unsupervised learning is discussed in Chapters 8 and 9.

We usually consider networks with separate inputs and outputs, and assume that we have a list or **training set** of correct input-output pairs as examples. When we apply one of the training inputs to the network we can compare the network output to the correct output, and then change the connection strengths w_{ij} to minimize the difference. This is typically done incrementally, making small adjustments in response to each training pair, so that the w_{ij}'s converge—if it works—to a solution in which the training set is "known" with high fidelity. It is then interesting to try input patterns *not* in the training set, to see whether the network can successfully **generalize** what it has learned.

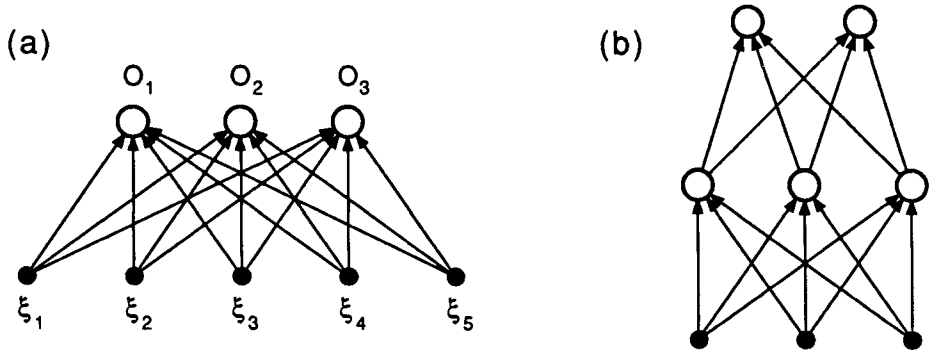

FIGURE 5.1 Perceptrons. (a) A simple perceptron which (by definition) has only one layer. (b) A two-layer perceptron. Inputs, shown as solid circles, perform no computation and are not included the count of layers.

5.1 Feed-Forward Networks

In this chapter and the next we study supervised learning in the context of a particular architecture: **layered feed-forward networks**. It is particularly simple to understand the learning process in this case. Note also that real neural structures in the brain are generally layered, and often largely (but not totally) feed-forward. We will turn to learning in more general networks in Chapter 7. Networks that are *not* strictly feed-forward, but include direct or indirect loops of connections, are often called **recurrent networks**. This includes the networks of Chapters 2 and 3 as well as those considered in Chapter 7.

Layered feed-forward networks were called **perceptrons** when first studied in detail by Rosenblatt and his coworkers 30 years ago [Rosenblatt, 1962]. Figure 5.1 shows two examples of perceptrons. There is a set of input terminals whose only role is to feed input patterns into the rest of the network. After this can come one or more intermediate layers of units, followed by a final output layer where the result of the computation is read off. In the restricted class of feed-forward networks under discussion, there are no connections leading from a unit to units in previous layers, nor to other units in the same layer, nor to units more than one layer ahead. Every unit (or input) feeds only the units in the next layer. The units in intermediate layers are often called **hidden units** because they have no direct connection to the outside world, neither input nor output.

There are two conventions in use for counting the number of layers in the network; some authors count the input terminals as a layer, some do not. We choose *not* to count them; we say for example that a network with one hidden layer is a two-layer network. This convention is becoming more frequently adopted, and seems more logical since the input "units" play no significant role. Note that an N-layer network has N layers of connections and $N-1$ hidden layers.

Feed-forward networks have by definition *asymmetric* connection matrices w_{ij}; all connections are unidirectional.[1] In general this means that there is no energy function; only with symmetric connections can the existence of an energy function be guaranteed. Thus we cannot employ *equilibrium* statistical mechanical methods here, for those rely on an energy function. We can however do some simple statistical calculations if we use stochastic units.

In this chapter we restrict ourselves further to *one-layer* feed-forward networks. These are often known as **simple perceptrons**. There is a set of N inputs, and an output layer, but no hidden layers. Figure 5.1(a) shows an example and defines our notation; the inputs and outputs are called ξ_k and O_i respectively. Its computation is simply described by

$$O_i = g(h_i) = g\Big(\sum_k w_{ik}\xi_k\Big) \tag{5.1}$$

where $g(h)$ is the activation function computed by the units. $g(h)$ is usually taken to be nonlinear; we can use a threshold function, a continuous sigmoid, or a stochastically determined ± 1. Particular cases will be discussed later.

Note that the output is an explicit function of the input. This is true for all feed-forward networks; the input is propagated through the network and produces the output right away. In contrast, recurrent networks always need some kind of relaxation to reach an attractor.

We have omitted any thresholds from our description because they can always be treated as connections to an input terminal that is permanently clamped at -1. Specifically we can fix $\xi_0 = -1$ and choose connections strengths $w_{i0} = \theta_i$ to obtain

$$O_i = g\Big(\sum_{k=0}^{N} w_{ik}\xi_k\Big) = g\Big(\sum_{k=1}^{N} w_{ik}\xi_k - \theta_i\Big) \tag{5.2}$$

with thresholds θ_i.[2]

The general association task can always be cast in the form of asking for a particular output pattern ζ_i^μ in response to an input pattern ξ_k^μ. That is, we want the *actual* output pattern O_i^μ to be equal to the **target pattern** ζ_i^μ

$$O_i^\mu = \zeta_i^\mu \qquad \text{(desired)} \tag{5.3}$$

for each i and μ. For the simple perceptron the actual output O_i^μ is given by (5.1) when the input ξ_k is clamped to the pattern ξ_k^μ:

$$O_i^\mu = g(h_i^\mu) = g\Big(\sum_k w_{ik}\xi_k^\mu\Big). \tag{5.4}$$

[1] Feed-forward networks are in general characterized by the possibility of numbering the units so that the weights w_{ij} form a triangular matrix, in which all entries above (or below) the diagonal are zero.

[2] If $g(h)$ is a continuous function "threshold" is not a very good term for θ. Often $-\theta$ is called a *bias* instead.

We define p as the number of input-output pairs in the training set, so $\mu = 1, 2, \ldots, p$.

The inputs, outputs, and targets may be boolean (e.g., ± 1) or continuous-valued. For the outputs this depends of course on the nature of the activation function $g(h)$. Sometimes we will use continuous-valued output units but have boolean targets, in which case we can only expect the outputs O_i^μ to come within some margin of the targets.

The general association task (5.3) includes our old associative memory problem as a special case; there we wanted the memory patterns ξ_k^μ to reproduce themselves when used as inputs. That is sometimes called **auto-association**, in contrast to the **hetero-association** task in which the output patterns ζ_i^μ are distinct from the input patterns ξ_k^μ. In feed-forward networks we will focus on hetero-association and hardly ever consider auto-association (see pages 132 and 136). For hetero-association the number of output units may be larger or smaller than the number of input units. Hetero-association includes **classification problems** where the inputs must be divided into particular output categories—normally only one output is on for each input—though these usually require more than a simple perceptron.

For simple perceptrons we will see that *if* there is a set of weights w_{ik} which achieves a particular computation, then these weights can be found by a simple learning rule. The learning rule starts from a general first guess at the weight values and then makes successive improvements. It actually reaches an appropriate answer in a *finite* number of steps.

There are, however, some rather simple and conceptually important computations which a one-layer network cannot do. We will examine in this chapter just what a one-layer network can and cannot do. In the next chapter we will see that multi-layer networks can solve many problems that are impossible within the one-layer architecture.

5.2 Threshold Units

We start with the simplest case of deterministic threshold units, $g(h) = \mathrm{sgn}(h)$, and assume that the targets ζ_i^μ also take ± 1 values. Then all that matters is the *sign* of the net input h_i^μ to output unit i; we want this sign to be the same as that of ζ_i^μ for each i and μ.

The output units are independent so it is often convenient to consider only one at a time and drop the i subscripts. Then the weights w_{ik} become a **weight vector** $\mathbf{w} = (w_1, w_2, \ldots, w_N)$ with one component for each input. Each input pattern ξ_k^μ can also be considered as a **pattern vector** $\boldsymbol{\xi}^\mu$ in this same N-dimensional space.

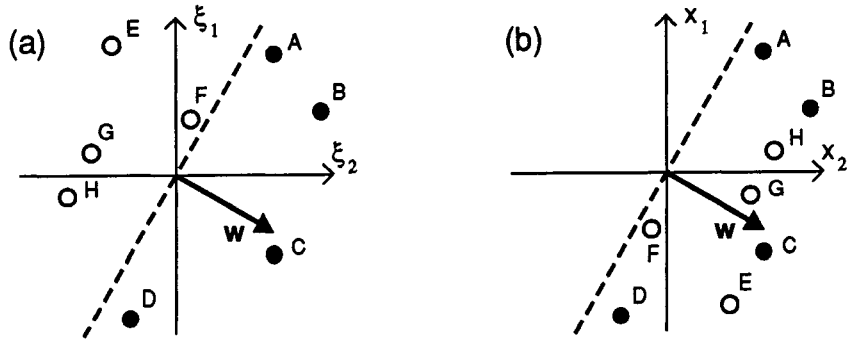

FIGURE 5.2 Geometrical illustration of the conditions (5.5) and (5.8). There are two continuous-valued inputs ξ_1 and ξ_2, and eight patterns ($\mu = 1 \ldots 8$) labelled A–H. Only one output is considered. The solid circles represent input patterns with $\zeta^\mu = +1$, whereas open circles mean $\zeta^\mu = -1$. In (a) the axes are ξ_1 and ξ_2 themselves, while in (b) they are x_1 and x_2; pattern μ has $\mathbf{x}^\mu = \zeta^\mu \boldsymbol{\xi}^\mu$. The condition for correct output is that the plane perpendicular to the weight vector $\mathbf{w}$ divides the points in (a), and lies on one side of all points in (b), as shown.

Then the condition (5.3) becomes[3]

$$\text{sgn}(\mathbf{w} \cdot \boldsymbol{\xi}_\mu) = \zeta^\mu \qquad \text{(desired)} \tag{5.5}$$

for every μ. This says that the weight vector $\mathbf{w}$ must be chosen so that the projection of pattern $\boldsymbol{\xi}^\mu$ onto it has the same sign as ζ^μ. But the boundary between positive and negative projections onto $\mathbf{w}$ is the plane[4] $\mathbf{w} \cdot \boldsymbol{\xi} = 0$ through the origin perpendicular to $\mathbf{w}$. So the condition for correct operation is that this plane should divide the inputs that have positive and negative targets, as illustrated in Fig. 5.2(a).

It is often convenient to use an alternate representation. By defining

$$x_k^\mu \equiv \zeta^\mu \xi_k^\mu \tag{5.6}$$

or

$$\mathbf{x}^\mu \equiv \zeta^\mu \boldsymbol{\xi}^\mu \tag{5.7}$$

we transform the condition (5.5) into

$$\mathbf{w} \cdot \mathbf{x}^\mu > 0 \qquad \text{(desired)} \tag{5.8}$$

[3]The scalar product $\mathbf{A}\cdot\mathbf{B}$ or $\mathbf{B}\cdot\mathbf{A}$ of two vectors $\mathbf{A}$ and $\mathbf{B}$ means $\sum_k A_k B_k$. It is equal to $|\mathbf{A}||\mathbf{B}|\cos\phi$, where ϕ is the angle between $\mathbf{A}$ and $\mathbf{B}$, and can be thought of as $|\mathbf{B}|$ times the projection of $\mathbf{A}$ onto $\mathbf{B}$, or vice versa. It is also known as the dot product or inner product.

[4]Or hyperplane. We often use the word *plane* generically; it means a line in two dimensions, an ordinary plane in three dimensions, and a hyperplane in four or more dimensions.

TABLE 5.1 AND function

ξ_1	ξ_2	ζ
0	0	−1
0	1	−1
1	0	−1
1	1	+1

for every μ. This says that the $\mathbf{x}$ vectors (each of which depends on one input-output pair from the training set) must all lie on the *same* side of the plane perpendicular to $\mathbf{w}$, as illustrated in Fig. 5.2(b).

Linear Separability

What happens if there is no such plane? Then the problem cannot be solved—the network cannot perform the task no matter how it is trained. So the condition for solvability of a problem by a simple perceptron with threshold units is whether or not that problem is **linearly separable**. A linearly separable problem is one in which a plane *can* be found in the $\boldsymbol{\xi}$ space separating the $\zeta^\mu = +1$ patterns from the $\zeta^\mu = -1$ ones. If there are several output units we must be able to find one such plane for *each* output.

If we have no threshold (or represent it implicitly by input ξ_0) the separating plane must go through the origin, as we have seen above. But it is interesting to reinstate an explicit threshold for a while. That turns the computation performed by the network into

$$O_i = \mathrm{sgn}\Big(\sum_{k>0} w_{ik}\xi_k - w_{i0}\Big) \tag{5.9}$$

or, for one unit in vector notation,

$$O = \mathrm{sgn}(\mathbf{w}\cdot\boldsymbol{\xi} - w_0)\,. \tag{5.10}$$

Thus the regions in the N-dimensional input space $(\xi_1, \xi_2, \ldots, \xi_N)$ with different decisions (± 1) for O are separated by an $(N-1)$-dimensional plane

$$\mathbf{w}\cdot\boldsymbol{\xi} = w_0 \tag{5.11}$$

in this space, distance w_0 from the origin. So the effect of adding an explicit threshold is simply to allow the separating plane not to go through the origin. Again there is one such plane for each output unit.

Some examples are appropriate. We consider first a simple example that *is* linearly separable: the Boolean **AND function**. It is a function of two binary 0/1 variables, so a perceptron with two input units ξ_1 and ξ_2 is needed. Because we are using the $\mathrm{sgn}(h)$ function the output is ± 1 (instead of 0/1), and we want to get a 1

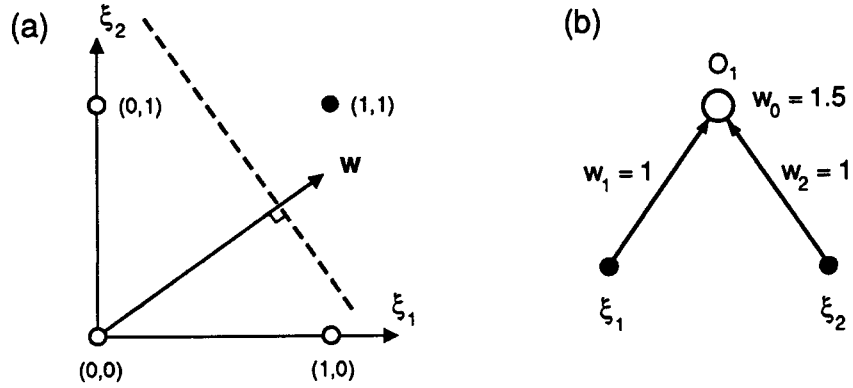

FIGURE 5.3 (a) The AND function is linearly separable. (b) A perceptron that implements AND.

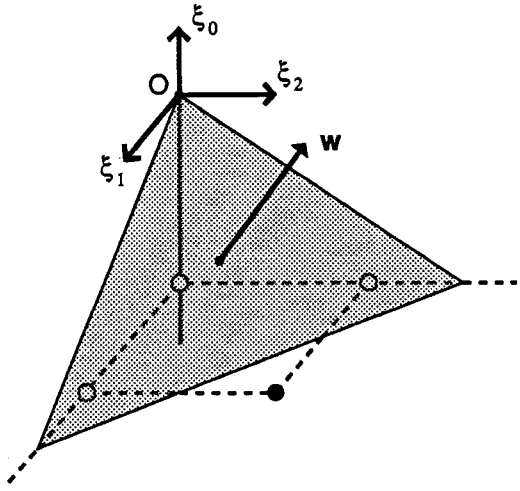

FIGURE 5.4 The AND problem with the threshold w_0 in an extra dimension ξ_0. Note that all the patterns have $\xi_0 = -1$ as in (5.2). The separating plane shown is perpendicular to the weight vector $\mathbf{w} = (1.5, 1, 1)$.

output only if both inputs are on. Table 5.1 lists the input-output pairs, or **truth table**, and Fig. 5.3(a) shows them in (ξ_1, ξ_2) space. It is easy to draw a line (a one-dimensional plane) to separate the "yes"-corner from the rest, so the problem is linearly separable and a simple perceptron can solve it. A suitable perceptron is shown in Fig. 5.3(b).

It is also interesting to consider the same problem with the threshold represented implicitly by weight w_0 to input $\xi_0 = -1$. Then the separating plane must go through the origin, but we have one extra dimension for ξ_0. Figure 5.4 shows the situation, with a separating plane drawn to correspond to the network of Fig. 5.3(b).

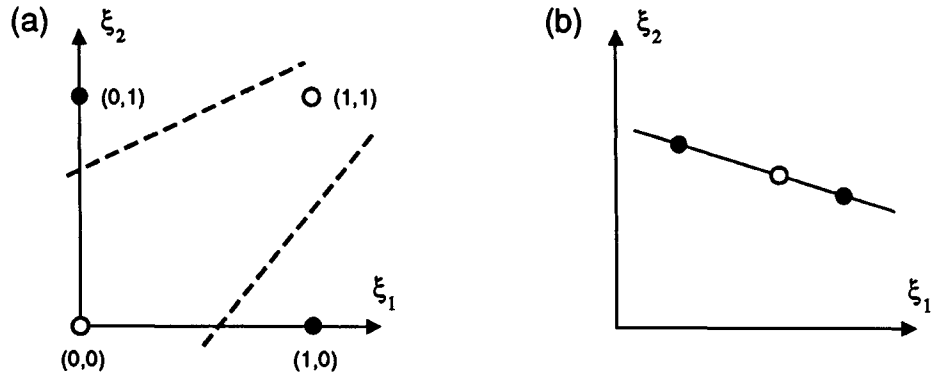

FIGURE 5.5 Problems that are *not* linearly separable. (a) The XOR problem. (b) Points that are not in "general position."

TABLE 5.2 XOR function

ξ_1	ξ_2	ζ
0	0	−1
0	1	+1
1	0	+1
1	1	−1

Now let us consider some examples that are *not* linearly separable. We return to representing thresholds explicitly, so separating planes need not be through the origin. Figure 5.5 shows two types of difficulties that can occur. In Fig. 5.5(a) there is clearly no plane (line) that can separate the two types of points. The same is true in (b), but only because the three points lie exactly in a straight line; if any of them were moved infinitesimally off the line a solution would be possible.

We discuss each case in a little more detail. The first, in Fig. 5.5(a), is the Boolean exclusive-OR or **XOR function**, with the truth table shown in table 5.2. The desired computation is a "yes" when one or the other of the inputs is on and a "no" when they are both off or both on. This is also the simplest case of the N-input **parity function** studied in detail by Minsky and Papert [1969].

The figure makes it clear that we cannot represent this function with a simple perceptron, but it is interesting to see the same thing algebraically. We just write out the equations (5.9) for the four patterns (suppressing the i index):

$$w_1 + w_2 < w_0 \tag{5.12}$$
$$-w_1 - w_2 < w_0 \tag{5.13}$$
$$w_1 - w_2 > w_0 \tag{5.14}$$
$$-w_1 + w_2 > w_0 . \tag{5.15}$$

Combining (5.12) and (5.15) we obtain $w_1 < 0$, while combining (5.13) and (5.14) we get the opposite, $w_1 > 0$. These cannot both be satisfied. Similarly, one finds impossible constraints on the other w_k's. Thus the network cannot do the computation.

Our other example, in Fig. 5.5(b), is very special. If, for example, we were choosing patterns from some continuous random distribution, there would be zero probability for three points to lie *exactly* on a line. Points are said to be **in general position** when this sort of special case does not occur. In an N-dimensional space a set of points is in general position if no more than $d + 1$ of them lie on any d-dimensional hyperplane, for any $d < N$. Without an explicit threshold the condition becomes that no more than d of them lie on any d-dimensional hyperplane *through the origin*, for any $d < N$; in two dimensions we'd need two points on a line through the origin to get into trouble. The no threshold condition is also equivalent to saying that all subsets of N or fewer points must be **linearly independent**.

We will show later that the first type of failure of linear separability, like XOR, can only occur when there are more patterns than inputs, $p > N$. On the other hand the second type can occur for any p.

A Simple Learning Algorithm

We now consider only linearly separable problems—so there is a solution—and ask how to find appropriate weights using a learning procedure. A simple procedure is to go through the input patterns one by one, and for each pattern go through the output units one by one, asking whether the output is the desired one ($O_i^\mu = \zeta_i^\mu$). If so, we leave the connections feeding into that unit alone. If not, then in the spirit of Hebb we add to each connection something proportional to the product of the input and the desired output. Specifically, we take

$$w_{ik}^{\text{new}} = w_{ik}^{\text{old}} + \Delta w_{ik} \tag{5.16}$$

where

$$\Delta w_{ik} = \begin{cases} 2\eta\zeta_i^\mu\xi_k^\mu & \text{if } \zeta_i^\mu \neq O_i^\mu ; \\ 0 & \text{otherwise;} \end{cases} \tag{5.17}$$

or

$$\Delta w_{ik} = \eta(1 - \zeta_i^\mu O_i^\mu)\zeta_i^\mu\xi_k^\mu \tag{5.18}$$

or

$$\Delta w_{ik} = \eta(\zeta_i^\mu - O_i^\mu)\xi_k^\mu . \tag{5.19}$$

The parameter η is called the **learning rate**. In (5.19) one can think of the two terms as learning the desired association and "unlearning" the erroneous one.

Instead of just asking that the *sign* of the input h_i^μ to the output units is correct (i.e., equal to ζ_i^μ), it is sometimes a good idea also to require that its *size* be larger than some margin:

$$\zeta_i^\mu h_i^\mu \equiv \zeta_i^\mu \sum_k w_{ik}\xi_k^\mu > N\kappa \qquad \text{(desired)}. \tag{5.20}$$

The sum on k scales with N, so to keep the **margin size** κ fixed for any number of input units we include an N on the right-hand side. We can implement this new criterion by changing (5.18) to

$$\Delta w_{ik} = \eta\,\Theta(N\kappa - \zeta_i^\mu h_i^\mu)\zeta_i^\mu \xi_k^\mu \tag{5.21}$$

where Θ is the unit step function (1.2)—we add the weight increment whenever (5.20) is not satisfied. The simpler learning rule (5.18) is just the special case with $\kappa = 0$ (apart from a factor of 2 in η).

Equation (5.21) is called the **perceptron learning rule** [Rosenblatt, 1962]. It can be proved [Block, 1962; Minsky and Papert, 1969] to converge to weights which accomplish the desired association (5.20) in a finite number of steps (provided of course that the solution exists). We provide a proof in the next section.

For a single output unit introducing κ just changes (5.8) into

$$\mathbf{w}\cdot\mathbf{x}^\mu > N\kappa \qquad \text{(desired)}. \tag{5.22}$$

This says geometrically that all points in $\mathbf{x}$-space must be further than $N\kappa/|\mathbf{w}|$ from the plane perpendicular to $\mathbf{w}$. In Fig. 5.2, for example, we can see that pattern F would fail to satisfy the condition unless κ were very small or $|\mathbf{w}|$ were very large.

We can also give a geometrical picture of the learning process. In our single-output vector notation, (5.21) becomes

$$\Delta\mathbf{w} = \eta\,\Theta(N\kappa - \mathbf{w}\cdot\mathbf{x}^\mu)\mathbf{x}^\mu \tag{5.23}$$

which says that the weight vector $\mathbf{w}$ is changed a little in the direction of $\mathbf{x}^\mu$ if its projection $\mathbf{w}\cdot\mathbf{x}^\mu$ onto $\mathbf{x}^\mu$ is less than $N\kappa/|\mathbf{w}|$. This is done over and over again until all projections are large enough. An example for $\kappa = 0$ is sketched in Fig. 5.6. Observe that a final solution was found after three steps $\mathbf{w} \to \mathbf{w}' \to \mathbf{w}'' \to \mathbf{w}'''$; there are no patterns left in the bad region of $\mathbf{w}'''$ so no further updates will occur.

Whatever the value of κ, a *direction* for $\mathbf{w}$ in which all the projections are positive will give a solution if scaled up to a large enough magnitude $|\mathbf{w}|$. Depending on the pattern vectors $\mathbf{x}^\mu$, there may be a wide range of such directions, or only a narrow cone, or (if no solution exists) none at all (Fig. 5.7 shows two examples). We can use this observation to quantify how easy or hard a problem is. The quantity

$$D(\mathbf{w}) = \frac{1}{|\mathbf{w}|}\min_\mu \mathbf{w}\cdot\mathbf{x}^\mu \tag{5.24}$$

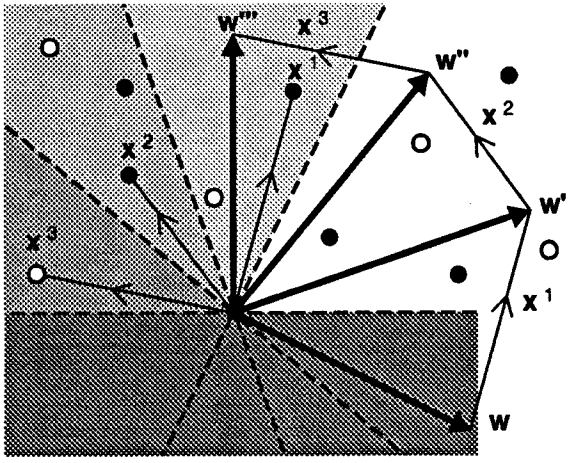

FIGURE 5.6 How the weight vector evolves during training for $\kappa = 0$, $\eta = 1$. Successive values of the weight vector are shown by $\mathbf{w}$, $\mathbf{w}'$, $\mathbf{w}''$, and $\mathbf{w}'''$. The darker and darker shading shows the "bad" region where $\mathbf{w} \cdot \mathbf{x} < 0$ for the successive $\mathbf{w}$ vectors. Each $\mathbf{w}$ is found from the previous one (e.g., $\mathbf{w}'$ from $\mathbf{w}$) by adding an $\mathbf{x}^\mu$ from the current bad region.

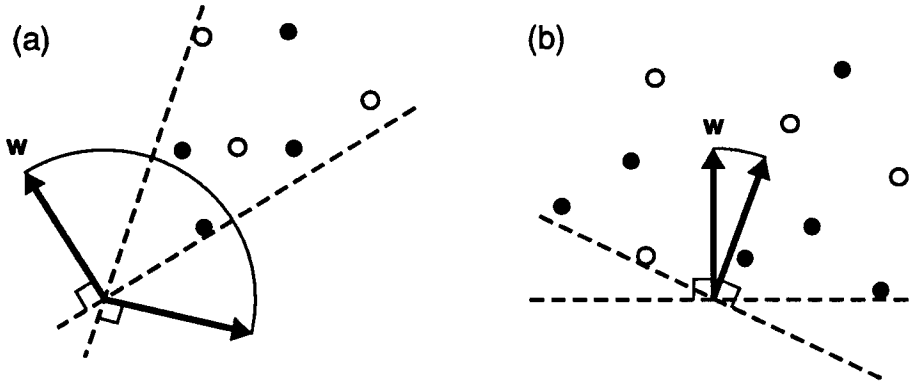

FIGURE 5.7 (a) An easy problem: any weight vector in the 135° angle shown will have positive pattern projections. (b) A harder problem where the weight vector must lie in a narrow cone.

depends on the *worst* of the projections. It is just the distance of the worst $\mathbf{x}^\mu$ to the plane perpendicular to $\mathbf{w}$, positive for the "good" side and negative for the "bad" side (see Fig. 5.8). The $1/|\mathbf{w}|$ factor makes it a function only of the direction of $\mathbf{w}$. If $D(\mathbf{w})$ is positive then all pattern points lie on the good side, and so a solution can be found for large enough $|\mathbf{w}|$.

If we maximize $D(\mathbf{w})$ over all possible weights we obtain the best direction for $\mathbf{w}$, along which a solution will be found for smallest $|\mathbf{w}|$. This solution is called the **optimal perceptron**. It can also be defined, equivalently, as the solution with largest margin size κ for fixed $|\mathbf{w}|$. The value

$$D_{\max} \equiv \max_{\mathbf{w}} D(\mathbf{w}) \tag{5.25}$$

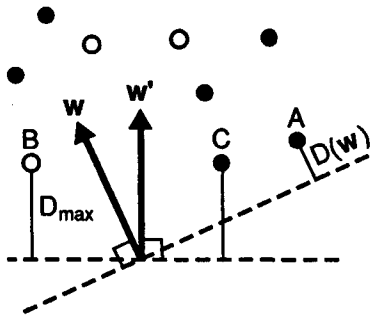

FIGURE 5.8 Definitions of $D(\mathbf{w})$ and $D_{\max}$. Pattern A is nearest to the plane perpendicular to weight vector $\mathbf{w}$, so the distance to A gives $D(\mathbf{w})$. Maximizing $D(\mathbf{w})$ with respect to $\mathbf{w}$ gives $\mathbf{w}'$, with $D(\mathbf{w}') = D_{\max}$. Note that both B and C are distance $D_{\max}$ from the plane.

of $D(\mathbf{w})$ in this direction tells us how easy the problem is: the larger $D_{\max}$, the easier the problem. If $D_{\max} < 0$, it cannot be solved. For example, $D_{\max}$ is $1/\sqrt{17}$ for the AND problem (from Fig. 5.4) and $-1/\sqrt{3}$ for XOR.

5.3 Proof of Convergence of the Perceptron Learning Rule ⋆

We assume that there *is* a solution to the problem and prove that the perceptron learning rule (5.21) reaches it in a *finite* number of steps. All we need to assume is that we can choose a weight vector $\mathbf{w}^*$ in a "good" direction; one in which $D(\mathbf{w}^*) > 0$. Our proof is related most closely to that given by Arbib [1987]; see also Rosenblatt [1962], Block [1962], Minsky and Papert [1969], and Diederich and Opper [1987].

At each step in the learning process a pattern is chosen and the weights are updated only if the condition (5.20) is *not* satisfied. Let M^μ denote the number of times that pattern μ has been used to update the weights at some point in the learning process. Then at that time

$$\mathbf{w} = \eta \sum_\mu M^\mu \mathbf{x}^\mu \tag{5.26}$$

if we assume that the initial weights are all zero.

The essence of the proof is to compute bounds on $|\mathbf{w}|$ and on the overlap $\mathbf{w}\cdot\mathbf{w}^*$ with our chosen "good" vector $\mathbf{w}^*$. These let us show that $\mathbf{w}\cdot\mathbf{w}^*/|\mathbf{w}|$ would get arbitrarily large if the total number of updates $M = \sum M^\mu$ kept on increasing. But this is impossible (since $\mathbf{w}^*$ is fixed), so the updating must cease at some finite M.

Consider $\mathbf{w}\cdot\mathbf{w}^*$ first. Using (5.26) and (5.24) we obtain

$$\mathbf{w}\cdot\mathbf{w}^* = \eta \sum_\mu M^\mu \mathbf{x}^\mu \cdot \mathbf{w}^* \geq \eta M \min_\mu \mathbf{x}^\mu \cdot \mathbf{w}^* = \eta M D(\mathbf{w}^*)|\mathbf{w}^*|. \tag{5.27}$$

Thus $\mathbf{w}\cdot\mathbf{w}^*$ grows like M. Now for an upper bound on $|\mathbf{w}|$, consider the change in length of $\mathbf{w}$ at a single update by pattern α:

$$\begin{aligned}\Delta|\mathbf{w}|^2 &= \left(\mathbf{w}+\eta\mathbf{x}^\alpha\right)^2 - \mathbf{w}^2 \\ &= \eta^2\left(\mathbf{x}^\alpha\right)^2 + 2\eta\mathbf{w}\cdot\mathbf{x}^\alpha \\ &\leq \eta^2 N + 2\eta N\kappa \\ &= N\eta(\eta+2\kappa)\,. \end{aligned} \tag{5.28}$$

The inequality comes directly from the condition $N\kappa \geq \mathbf{w}\cdot\mathbf{x}^\alpha$ for performing an update with pattern α. Note that we also used $x_k^\alpha = \pm 1$, so that $(\mathbf{x}^\alpha)^2 = N$, but the proof is easily generalized to other types of patterns. By summing the increments to $|\mathbf{w}|^2$ for M steps we obtain the desired bound

$$|\mathbf{w}|^2 \leq MN\eta(\eta+2\kappa)\,. \tag{5.29}$$

Thus $|\mathbf{w}|$ grows no faster than $\sqrt{M}$, and therefore from (5.27) the ratio $\mathbf{w}\cdot\mathbf{w}^*/|\mathbf{w}|$ grows at least as fast as $\sqrt{M}$. But this cannot continue, so M must stop growing.

More precisely, we can bound the normalized scalar product

$$\phi = \frac{(\mathbf{w}\cdot\mathbf{w}^*)^2}{|\mathbf{w}|^2|\mathbf{w}^*|^2} \tag{5.30}$$

which is the squared cosine of the angle between $\mathbf{w}$ and $\mathbf{w}^*$. Because it is the squared cosine it is obviously less than or equal to 1 (as also follows from the Cauchy-Schwarz inequality). But with (5.27) and (5.29) we find

$$1 \geq \phi \geq M\frac{D(\mathbf{w}^*)^2\eta}{N(\eta+2\kappa)} \tag{5.31}$$

which gives us an upper bound on the number of weight updates (using the best possible $\mathbf{w}^*$):

$$M \leq N\frac{1+2\kappa/\eta}{D_{\max}^2}\,. \tag{5.32}$$

This bound is proportional to the number of input units, but interestingly enough it does *not* depend on the number of patterns p. Of course the real convergence time does depend on p because one typically has to continue checking all p patterns to find the ones for which a weight update is needed; the number of such checks increases with p even if the number of actual updates does not. Additionally, $D_{\max}$ typically decreases with increasing p, resulting in a growing M. Note also that the bound on M grows linearly with κ, because for larger κ the learning must reach a larger $|\mathbf{w}|$ along any given good direction.

5.4 Linear Units

So far in our study of simple perceptrons we have considered only threshold units, with $g(h) = \mathrm{sgn}(h)$. We turn now to continuous-valued units, with $g(h)$ a continuous and *differentiable* function of u. The great advantage of such units is that they allow us to construct a cost function $E[\mathbf{w}]$ which measures the system's performance error as a differentiable function of the weights $\mathbf{w} = \{w_{ik}\}$. We can then use an optimization technique, such as gradient descent, to minimize this error measure.

We start in this section with **linear units**, for which $g(h) = h$. These are not as useful practically as the nonlinear networks considered next, but are simpler and allow more detailed analysis. The output of a linear simple perceptron subjected to an input pattern ξ_k^μ is given by

$$O_i^\mu = \sum_k w_{ik}\xi_k^\mu \tag{5.33}$$

and the desired association is $O_i^\mu = \zeta_i^\mu$ as usual, or

$$\zeta_i^\mu = \sum_k w_{ik}\xi_k^\mu \qquad \text{(desired)}. \tag{5.34}$$

Note that the O_i^μ's are continuous-valued quantities now, although we could still restrict the desired values ζ_i^μ to ± 1.

Explicit Solution

For a linear network we can actually compute a suitable set of weights explicitly using the **pseudo-inverse** method. We saw on page 50 how to find the weights to satisfy (5.34) for the special case of auto-association, $\zeta_i^\mu = \xi_i^\mu$. The generalization to hetero-association is just

$$w_{ik} = \frac{1}{N}\sum_{\mu\nu} \zeta_i^\mu (\mathbf{Q}^{-1})_{\mu\nu}\xi_k^\nu \tag{5.35}$$

where

$$\mathbf{Q}_{\mu\nu} = \frac{1}{N}\sum_k \xi_k^\mu \xi_k^\nu \tag{5.36}$$

is the overlap matrix of the *input* patterns. It is straightforward to check that (5.35) solves (5.34), as in (3.19).

Observe that (5.35) only applies if $\mathbf{Q}^{-1}$ exists, and that this condition only depends on the *input* patterns. It requires the input patterns to be **linearly independent**. If on the contrary there exists a linear relationship

$$a_1\xi_k^1 + a_2\xi_k^2 + \cdots + a_p\xi_k^p = 0 \qquad \text{(for all } k\text{)} \tag{5.37}$$

between them, then the outputs O_i^μ cannot be independently chosen and the problem is normally insoluble.

Linear independence is the general condition for solvability in the linear network. It is actually a sufficient, but not a necessary, condition; even with linearly dependent input patterns it *might* happen that the targets were such that a solution could be found, though not by the present method. But a very special choice would be needed.

A set of p input patterns can only be linearly independent if $p \leq N$, so we can store at most N arbitrary associations in a linear network. But a set of N or fewer patterns is not necessarily linearly independent. If a dependency exists, then the input pattern vectors $\xi_k^1, \xi_k^2, \ldots, \xi_k^p$ only span a **pattern subspace** of the input space, and the solution (5.35) is not unique. If w_{ik} is one solution, and ξ_k^* is any vector orthogonal to the subspace of the input patterns, so $\sum_k \xi_k^\mu \xi_k^* = 0$ for all μ, then the weights w'_{ik} given by

$$w'_{ik} = w_{ik} + a_i \xi_k^* \tag{5.38}$$

for any a_i provide another solution.

The linear independence condition for linear (and nonlinear) units is quite distinct from the linear separability condition found for threshold units. Linear independence does imply linear separability, but the reverse is not true. In fact most of the problems of interest in threshold networks do *not* satisfy the linear independence condition, because they typically have $p > N$. This includes AND and XOR, and indeed all the other threshold network examples used or illustrated in this chapter.

Gradient Descent Learning

We could use (5.35) and (5.36) to compute a set of weights w_{ik} that produce exactly the desired outputs from each input pattern. But we are more interested here in finding a *learning* rule that allows us to find such a set of weights by successive improvement from an arbitrary starting point.

We define an error measure or **cost function** by

$$E[\mathbf{w}] = \frac{1}{2}\sum_{i\mu}(\zeta_i^\mu - O_i^\mu)^2 = \frac{1}{2}\sum_{i\mu}\Big(\zeta_i^\mu - \sum_k w_{ik}\xi_k^\mu\Big)^2 . \tag{5.39}$$

This is smaller the better our w_{ik}'s are; E is normally positive, but goes to zero as we approach a solution satisfying (5.34). Note that this cost function depends only on the weights w_{ik} and the problem patterns. In contrast, the energy functions considered in Chapters 2 to 4 depended on the current state of the network, which evolved so as to minimize the energy. Here the evolution is of the weights (learning), not of the activations of the units themselves.

Given our error measure $E[\mathbf{w}]$, we can improve on a set of w_{ik}'s by sliding downhill on the surface it defines in $\mathbf{w}$ space. Specifically, the usual **gradient descent algorithm** suggests changing each w_{ik} by an amount Δw_{ik} proportional to the gradient of E at the present location:

$$\begin{aligned}\Delta w_{ik} &= -\eta \frac{\partial E}{\partial w_{ik}} \\ &= \eta \sum_{\mu} (\zeta_i^\mu - O_i^\mu)\xi_k^\mu \,. \end{aligned} \tag{5.40}$$

If we make these changes individually for each input pattern ξ_k^μ in turn, we have simply

$$\Delta w_{ik} = \eta(\zeta_i^\mu - O_i^\mu)\xi_k^\mu \tag{5.41}$$

for the changes in response to pattern μ, or

$$\Delta w_{ik} = \eta \delta_i^\mu \xi_k^\mu \tag{5.42}$$

if we define the errors (or **deltas**) δ_i^μ by

$$\delta_i^\mu = \zeta_i^\mu - O_i^\mu \,. \tag{5.43}$$

This result—(5.41), or (5.42) plus (5.43)—is commonly referred to as the **delta rule**, or the **adaline rule**, or the **Widrow-Hoff rule**, or the **LMS** (least mean square) **rule** [Rumelhart, McClelland, et al., 1986; Widrow and Hoff, 1960]. It is also nearly identical to the Rescorla-Wagner model of classical conditioning in behavioral psychology [Rescorla and Wagner, 1972].

Equation (5.41) is identical to our simple rule (5.19) for the threshold unit network. Note however that (5.19) originated in an empirical Hebb assumption, but now we have derived it from gradient descent. As we will see, the new approach is easily generalized to more layers, whereas the Hebb rule is not. The new approach obviously requires continuous-valued units with differentiable activation functions.

Actually the equivalence of (5.41) and (5.19) is a little deceptive because O_i^μ is a different function of the inputs. In the threshold case the term $\zeta_i^\mu - O_i^\mu$ is either 0 or ± 2, whereas here it can take any value. As a result the final weight values are *not* the same in the two cases. Nor are the conditions for existence of a solution (linear separability versus linear independence). And in the threshold case the learning rule stops after a finite number of steps, while here it continues in principle for ever, converging only asymptotically to the solution.

The cost function (5.39) is just a quadratic form in the weights. In the subspace spanned by the patterns the surface is a parabolic bowl with a single minimum. Assuming that the pattern vectors are linearly independent, so that there *is* a solution to (5.34), the minimum is at $E = 0$. In the directions (if any) of $\mathbf{w}$-space orthogonal to all the pattern vectors the error is constant, as may be seen by inserting (5.38) into (5.39); the ξ_k^* term makes no difference because $\sum_k \xi_k^\mu \xi_k^* = 0$.

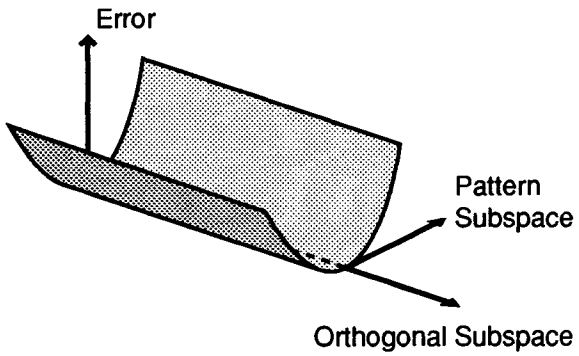

FIGURE 5.9 The "rain gutter" shape of the error surface for linear units. The error takes its minimum value of 0 along level valleys if the patterns ξ_k^μ do not span the whole of $\boldsymbol{\xi}$-space.

In other words, the error surface in weight space is like a **rain gutter**, as pictured in Fig. 5.9, with infinite level valleys in the directions orthogonal to the pattern vectors.

The gradient descent rule (5.40) or (5.41) produces changes in the weight vectors $\mathbf{w}_i = (w_{i1}, \ldots, w_{iN})$ *only* in the directions of the pattern vectors $\boldsymbol{\xi}^\mu$. Thus any component of the weights orthogonal to the patterns is left unchanged by the learning, leaving an inconsequential uncertainty of exactly the form (5.38) in the final solution. Within the pattern subspace the gradient descent rule necessarily decreases the error if η is small enough, because it takes us in the downhill gradient direction. Thus with enough iterations we approach the bottom of the valley arbitrarily closely, from any starting point. And any point at the bottom of the valley solves the original problem (5.34) exactly.

Convergence of Gradient Descent ⋆

The argument just given for convergence to the bottom of the valley is intuitively reasonable but bears a little further analysis. The first step is to diagonalize the quadratic form (5.39). As long as the pattern vectors are linearly independent, this allows us to write $E[\mathbf{w}]$ in the form

$$E = \sum_{\lambda=1}^{M} a_\lambda \left(w_\lambda - w_\lambda^0\right)^2 . \tag{5.44}$$

Here M is the total number of weights, equal to N times the number of output units, and the w_λ's are linear combinations of the w_{ik}'s. The a_λ's and w_λ^0's are constants depending only on the pattern vectors. The eigenvalues a_λ are necessarily positive or zero because of the sum-of-squares form of (5.39); the quadratic form is positive semi-definite.

Notice first that if some of the a_λ's are zero then E is independent of the corresponding w_λ's. This is equivalent to the rain gutters already described; the

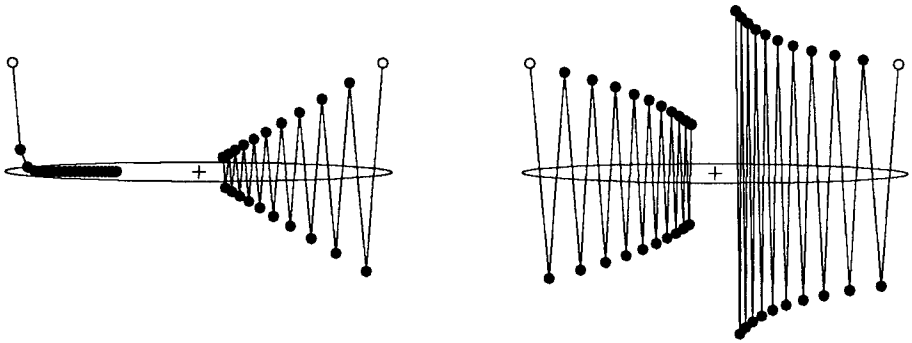

FIGURE 5.10 Gradient descent on a simple quadratic surface (the left and right parts are copies of the same surface). Four trajectories are shown, each for 20 steps from the open circle. The minimum is at the + and the ellipse shows a constant error contour. The only significant difference between the trajectories is the value of η, which was 0.02, 0.0476, 0.049, and 0.0505 from left to right.

corresponding eigenvector directions in **w** space are orthogonal to all the pattern vectors.

Now let us perform gradient descent on (5.44). Because the transformation from w_{ik} to w_λ is linear, this is entirely equivalent to gradient descent in the original basis. But the diagonal basis makes it much easier:

$$\Delta w_\lambda = -\eta \frac{\partial E}{\partial w_\lambda} = -2\eta a_\lambda (w_\lambda - w_\lambda^0) . \tag{5.45}$$

Thus the distance $\delta w_\lambda = w_\lambda - w_\lambda^0$ from the optimum in the λ-direction is transformed according to

$$\delta w_\lambda^{\text{new}} = \delta w_\lambda^{\text{old}} + \Delta w_\lambda = (1 - 2\eta a_\lambda)\delta w_\lambda^{\text{old}} . \tag{5.46}$$

In the directions for which $a_\lambda > 0$ we therefore get closer to the optimum as long as $|1 - 2\eta a_\lambda| < 1$. The approach is first order; each distance δw_λ gets multiplied by a fixed factor at each iteration. The value of η is limited by the largest eigenvalue $a_\lambda^{\max}$, corresponding to the steepest curvature direction of the error surface; we must have $\eta < 1/a_\lambda^{\max}$ or we will end up jumping too far, further up the other side of the valley than we are at present. But the rate of approach to the optimum is usually limited by the *smallest* non-zero eigenvalue $a_\lambda^{\min}$, corresponding to the shallowest curvature direction. If $a_\lambda^{\max}/a_\lambda^{\min}$ is large progress along the shallow directions can be excruciatingly slow.

Figure 5.10 illustrates these points in a simple case. We show gradient descent on the surface $E = x^2 + 20y^2$ for 20 iterations at different values of η. This quadratic form is already diagonal, with $a_1 = 1$ and $a_2 = 20$. Notice the distance of the last point from the minimum. At $\eta = 0.02$ we reach $y \approx 0$ fairly quickly, but then make

only slow progress in x. At the other extreme, if $\eta > 1/20 = 0.05$ the algorithm produces a divergent oscillation in y. The fastest approach is approximately when the x and y multipliers $|1 - 2\eta|$ and $|1 - 40\eta|$ are equal, giving $\eta = 1/21 = 0.476$ (second illustration).

Of course this analysis assumes that we actually take steps along the gradient. Progress will normally be somewhat slower if instead we change one component at a time, as in (5.41), though the incremental algorithm is often more convenient in practice. Another alternative sometimes used is to take steps of constant *size* (usually decreasing with time) in the gradient *direction*; this can help to speed up convergence when $a_\lambda^{\max}/a_\lambda^{\min}$ is large, but requires careful control.

We have also assumed of course that a perfect solution exists, which requires linear independence of the patterns. It is however interesting to ask what happens if the patterns are *not* linearly independent. Then we can only diagonalize (5.39) in the form

$$E = E_0 + \sum_{\lambda=1}^{M} a_\lambda \left(w_\lambda - w_\lambda^0\right)^2 \tag{5.47}$$

with $E_0 > 0$. There is still a single minimum (or gutter), but $E = E_0 > 0$ there, showing that the desired association has not been found. Trying gradient descent on the XOR problem, for example, produces a single minimum with $E = 2$ at $w_0 = w_1 = w_2 = 0$, which makes the output always 0. The XOR problem is obviously not linearly independent, since $p > N$.

5.5 Nonlinear Units

It is straightforward to generalize gradient descent learning from the linear $g(h) = h$ discussed in the previous section to networks with any differentiable $g(h)$. The sum-of-squares cost function becomes

$$E[\mathbf{w}] = \frac{1}{2}\sum_{i\mu}(\zeta_i^\mu - O_i^\mu)^2 = \frac{1}{2}\sum_{i\mu}\Big[\zeta_i^\mu - g\Big(\sum_k w_{ik}\xi_k^\mu\Big)\Big]^2 . \tag{5.48}$$

We then find

$$\frac{\partial E}{\partial w_{ik}} = -\sum_\mu [\zeta_i^\mu - g(h_i^\mu)]g'(h_i^\mu)\xi_k^\mu \tag{5.49}$$

so the gradient descent correction $-\eta\partial E/\partial w_{ik}$ to w_{ik} after presentation with pattern number μ is of the same form as (5.42):

$$\Delta w_{ik} = \eta\delta_i^\mu \xi_k^\mu . \tag{5.50}$$

But now the quantity

$$\delta_i^\mu = [\zeta_i^\mu - O_i^\mu]g'(h_i^\mu) \tag{5.51}$$

has acquired an extra factor $g'(h_i^\mu)$ of the derivative of the activation function $g(h)$. With a sigmoid form for $g(h)$, such as $g(h) = \tanh(h)$, this derivative is largest when $|h_i^\mu|$ is small. Thus the changes are made most strongly on connections feeding into units with small $|h_i^\mu|$'s, those which are "in doubt" about their output.

We remark that a hyperbolic tangent function $g(h) = \tanh(\beta h)$ is particularly convenient because its derivative is given by $g'(h) = \beta(1-g^2)$. Thus one doesn't have to recompute the derivative of g in (5.51) once one has found $O_i^\mu = g(h_i^\mu)$ itself. The same applies of course to the corresponding choice of the sigmoid function $g(h) = f_\beta(h) = [1 + \exp(-2\beta h)]^{-1}$ for units with outputs between 0 and 1, for which $g'(h) = 2\beta g(1-g)$.

The conditions for the existence of a solution are exactly the same as for the linear case: linear independence of the patterns. This is because the solution to our present problem is equivalent to the linear one with the targets ζ_i^μ replaced by $g^{-1}(\zeta_i^\mu)$.[5] But the question of whether gradient descent *finds* the solution, assuming it exists, is not the same as for the linear case. If the targets lie outside the range of $g(h)$ (e.g., ± 1 targets with $g(h) = \tanh h$), it is possible in the nonlinear case for the cost function to have **local minima** besides the global minimum at $E = 0$. The gradient descent algorithm can then become stuck in such a local minimum.

Nonlinear units do not allow a perfect solution of problems that are not linearly independent, but they may help by offering alternate **partial solutions**. If, for example, we try the XOR problem with a $\tanh(\beta h)$ activation function, we find five possible minima, each with $E = 2$. There is the one at $w_0 = w_1 = w_2 = 0$ (which makes the output always 0) that the linear network found. But now there are four more at $|w_0| = |w_1| = |w_2| \to \infty$ with 1 or 3 negative signs; these each produce the correct output for three out of the four input patterns, but get the wrong sign on the fourth. It is arguable that three out of four is better than four "don't know's". However the convergence is very slow for large β because most of the landscape is rather flat.

In simple perceptrons the main advantage of nonlinear activation functions is that they can keep the output between fixed bounds, such as ± 1 for a $\tanh(h)$ function. They are much more important in multi-layer networks where they make possible the solution of problems that are *not* possible with linear units. A multi-layer linear feed-forward network is exactly equivalent to a one-layer one in the computation it performs (since a linear transformation of a linear transformation is a linear transformation), so such a network has the same limitations as a one-layer one. In particular it can only work if the input patterns are linearly independent. But this restriction is *not* present for a multi-layer *nonlinear* feed-forward network.

Other Cost Functions

The quadratic cost function (5.48) is not the only possibility. Any differentiable

[5] We normally consider only monotonic activation functions, which are always invertible except at the endpoints.

function of ζ_i^μ and O_i^μ that is minimized by $O_i^\mu = \zeta_i^\mu$ could be used. The gradient descent rule $\Delta w_{ik} \propto -\partial E/\partial w_{ik}$ gives a prescription analogous to (5.50) for each such choice.

The choice

$$E = \sum_{i\mu} \left[\tfrac{1}{2}(1+\zeta_i^\mu) \log \frac{1+\zeta_i^\mu}{1+O_i^\mu} + \tfrac{1}{2}(1-\zeta_i^\mu) \log \frac{1-\zeta_i^\mu}{1-O_i^\mu} \right] \tag{5.52}$$

has received particular attention [Baum and Wilczek, 1988; Hopfield, 1987; Solla et al., 1988]. It has a natural interpretation in terms of learning the correct probabilities of a set of hypotheses represented by the output units, using $\frac{1}{2}(1+O_i^\mu)$ for the probability that the hypothesis represented by unit i is true: $O_i^\mu = -1$ means definitely false, and $O_i^\mu = +1$ means definitely true. Similarly $\frac{1}{2}(1+\zeta_i^\mu)$ is interpreted as a target set of probabilities. Then information theory suggests the **relative entropy** (5.52) of these probability distributions as a natural measure of the difference between them [Kullback, 1959].

Like (5.48), (5.52) is always positive except when $O_i^\mu = \zeta_i^\mu$ for all i and μ, where $E = 0$. Its advantage is, qualitatively, that it diverges if the output of one unit saturates at the wrong extreme. The quadratic measure (5.48) just approaches a constant in that case, and therefore the learning can float around on a relatively flat plateau of E for a long time. The use of the entropy measure has been shown [Wittner and Denker, 1988] to solve some learning problems that cannot be solved using the quadratic E. It is also appropriate if the training set data are actually probabilistic or fuzzy as, for instance, in the association of symptoms with causes in medical diagnosis.

Differentiating (5.52) and taking g to be a tanh gives the same change of weights as (5.50) but with

$$\delta_i^\mu = \beta(\zeta_i^\mu - O_i^\mu) . \tag{5.53}$$

The only difference (besides a factor of β) from (5.51) is that the $g'(h_i^\mu)$ factor is missing. The result is essentially identical to the delta rules found in the linear network (5.43) and in the threshold network (5.19).

Gradient descent learning is sometimes used for **binary decision problems** where the network is trained with, say, ± 1 targets, but then used with any positive output $O_i > 0$ being taken as a "yes" and any negative output $O_i < 0$ as a "no". This can sometimes produce satisfactory results on problems which are linearly separable but not linearly independent. However gradient descent with the usual quadratic cost function does not necessarily find a viable solution in such cases, even though the perceptron learning rule *would* work. But Wittner and Denker [1988] have shown that a class of alternative **well-formed** cost functions do work in such situations, so that gradient descent always finds a solution if there is one. They define a well-formed cost function so that the magnitude of its gradient is always larger than a constant, not simply greater than zero, whenever an output has the wrong sign. The entropic measure (5.52) is well formed in this sense.

5.6 Stochastic Units

Another generalization is from our deterministic units to **stochastic units** S_i governed by (2.48):

$$\text{Prob}(S_i^\mu = \pm 1) = \frac{1}{1+\exp(\mp 2\beta h_i^\mu)} \tag{5.54}$$

with

$$h_i^\mu = \sum_k w_{ik}\xi_k^\mu \tag{5.55}$$

as before. This leads to

$$\langle S_i^\mu \rangle = \tanh\Big(\beta \sum_k w_{ik}\xi_k^\mu\Big) \tag{5.56}$$

just as in (2.42). In the context of a simulation we can use (5.56) to *calculate* $\langle S_i^\mu \rangle$, whereas in a real stochastic network we would find it by averaging S_i for a while, updating randomly chosen units according to (5.54). Either way, we then use $\langle S_i^\mu \rangle$ as the basis of a weight change

$$\Delta w_{ik} = \eta \delta_i^\mu \xi_k^\mu \tag{5.57}$$

where

$$\delta_i^\mu = \zeta_i^\mu - \langle S_i^\mu \rangle . \tag{5.58}$$

This is just the average over outcomes of the changes we would have made on the basis of individual outcomes using the ordinary delta rule (5.43). We will find it particularly important when we discuss reinforcement learning in Section 7.4.

It is interesting to prove that this rule always decreases the average error given by the usual quadratic measure

$$E = \frac{1}{2}\sum_{i\mu}(\zeta_i^\mu - S_i^\mu)^2 . \tag{5.59}$$

Since we are assuming output units and patterns which are ± 1, this is just twice the total number of bits in error, and can also be written

$$E = \sum_{i\mu}(1 - \zeta_i^\mu S_i^\mu) . \tag{5.60}$$

Thus the *average* error in the stochastic network is

$$\begin{aligned} \langle E \rangle &= \sum_{i\mu}(1 - \zeta_i^\mu \langle S_i^\mu \rangle) \\ &= \sum_{i\mu}\Big[1 - \zeta_i^\mu \tanh\Big(\beta \sum_k w_{ik}\xi_k^\mu\Big)\Big] . \end{aligned} \tag{5.61}$$

The change in $\langle E \rangle$ in one cycle of weight updatings is thus

$$\begin{aligned}\Delta\langle E\rangle &= \sum_{ik} \frac{\partial\langle E\rangle}{\partial w_{ik}} \Delta w_{ik} \\ &= -\sum_{i\mu k} \Delta w_{ik} \zeta_i^\mu \frac{\partial}{\partial w_{ik}} \tanh(\beta h_i^\mu) \\ &= -\sum_{i\mu k} \eta[1 - \zeta_i^\mu \tanh(\beta h_i^\mu)]\beta \mathrm{sech}^2(\beta h_i^\mu) \end{aligned} \tag{5.62}$$

using[6] $d\tanh(x)/dx = \mathrm{sech}^2 x$. The result (5.62) is clearly always negative (recall $\tanh(x) < 1$), so the procedure always improves the average performance.

5.7 Capacity of the Simple Perceptron ⋆

In the case of the associative network in Chapter 2 we were able to find the **capacity** $p_{\max}$ of a network of N units; for random patterns we found $p_{\max} = 0.138N$ for large N if we used the standard Hebb rule. If we tried to store p patterns with $p > p_{\max}$ the performance became terrible.

Similar questions can be asked for simple perceptrons:

- How many *random* input-output pairs can we expect to store reliably in a network of given size?
- How many of these can we expect to *learn* using a particular learning rule?

The answer to the second question may well be smaller than the first (e.g., for nonlinear units), but is presently unknown in general. The first question, which this section deals with, gives the maximum capacity that *any* learning algorithm can hope to achieve.

For continuous-valued units (linear or nonlinear) we already know the answer, because the condition is simply linear independence. If we choose p *random* patterns, then they will be linearly independent if $p \leq N$ (except for cases with very small probability). So the capacity is $p_{\max} = N$.

The case of threshold units depends on linear separability, which is harder to deal with. The answer for random continuous-valued inputs was derived by Cover [1965] (see also Mitchison and Durbin [1989]) and is remarkably simple:

$$p_{\max} = 2N. \tag{5.63}$$

As usual N is the number of *input* units, and is presumed large. The number of *output* units must be small and fixed (independent of N). Equation (5.63) is strictly true in the $N \to \infty$ limit.

[6] The function $\mathrm{sech}^2 x = 1 - \tanh^2 x$ is a bell-shaped curve with peak at $x = 0$.

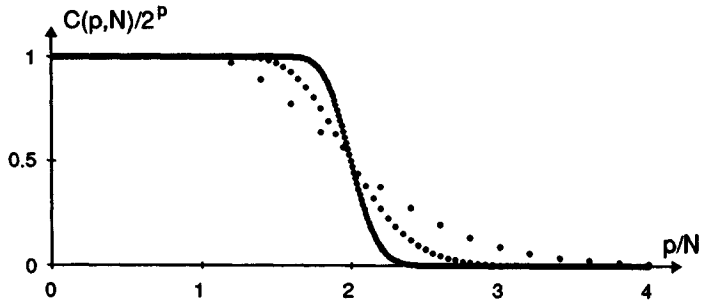

FIGURE 5.11 The function $C(p, N)/2^p$ given by (5.67) plotted versus p/N for $N = 5$, 20, and 100.

The rest of this section is concerned with proving (5.63), and may be omitted on first reading. We follow the approach of Cover [1965]. A more general (but much more difficult) method for answering this sort of question was given by Gardner [1987] and is discussed in Chapter 10.

We consider a perceptron with N continuous-valued inputs and one ± 1 output unit, using the deterministic threshold limit. The extension to several output units is trivial since output units and their connections are independent—the result (5.63) applies separately to each. For convenience we take the thresholds to be zero, but they could be reinserted at the expense of one extra input unit, as in (5.2).

In (5.11) we showed that the perceptron divides the N-dimensional input space into two regions separated by an $(N-1)$-dimensional hyperplane. For the case of zero threshold this plane goes through the origin. All the points on one side give an output of $+1$ and all those on the other side give -1. Let us think of these as red $(+1)$ and black (-1) points respectively. Then the question we need to answer is: how many points can we expect to put randomly in an N-dimensional space, some red and some black, and then find a hyperplane through the origin that divides the red points from the black points?

Let us consider a slightly different question. For a given set of p randomly placed points in an N-dimensional space, for how many out of the 2^p possible red and black colorings of the points can we find a hyperplane dividing red from black? Call the answer $C(p, N)$. For p small we expect $C(p, N) = 2^p$, because we should be able to find a suitable hyperplane for *any* possible coloring; consider $N = p = 2$ for example. For p large we expect $C(p, N)$ to drop well below 2^p, so an arbitrarily chosen coloring will *not* possess a dividing hyperplane. The transition between these regimes turns out to be sharp for large N, and gives us $p_{\max}$.

We will calculate $C(p, N)$ shortly, but let us first examine the result. Figure 5.11 shows a graph of $C(p, N)/2^p$ against p/N for $N = 5$, 20, and 100. Our expectations for small and large p are fulfilled, and we see that the transition occurs quite rapidly in the neighborhood of $p = 2N$, in agreement with (5.63). As N is made larger and

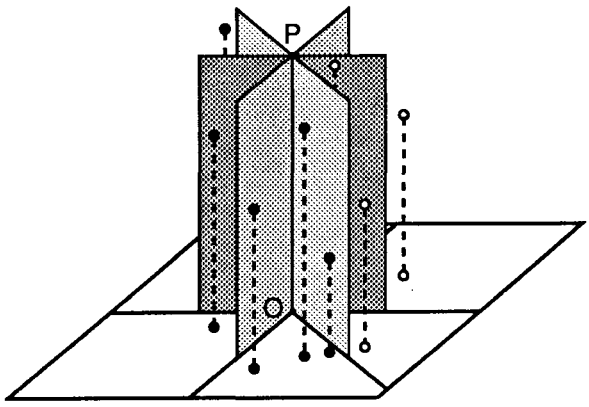

FIGURE 5.12 Finding separating hyperplanes constrained to go through a point P as well as the origin O is equivalent to projecting onto one lower dimension.

larger the transition becomes more and more sharp. Thus (5.63) is justified if we can demonstrate that Fig. 5.11 is correct.

The random placement of points is not actually necessary.[7] All that we need is that the points be in **general position**. As discussed on page 97, this means (for the no threshold case) that all subsets of N (or fewer) points must be linearly independent. As an example consider $N = 2$: a set of p points in a two-dimensional plane is in general position if no two lie on the same line through the origin. A set of points chosen from a continuous random distribution will obviously be in general position except for coincidences that have zero probability.

We can now calculate $C(p, N)$ by induction. Let us call a coloring that *can* be divided by a hyperplane a **dichotomy**. Suppose we start with p points and add a new point P. Then the $C(p, N)$ old dichotomies fall into two classes:

- For those previous dichotomies where the dividing hyperplane could have been drawn *through* point P, there'll be *two* new dichotomies, one with P red and one with it black. This is because when the points are in general position any hyperplane through P can be shifted infinitesimally to go either side of it, without changing the side of any of the other p points.
- For the remainder of the previous dichotomies only one color of point P will fit, so there'll be *one* new dichotomy for each old one.

Thus

$$C(p+1, N) = C(p, N) + D \tag{5.64}$$

where D is the number of the previous $C(p, N)$ dichotomies that could have had the dividing hyperplane drawn through P as well as the origin O. But this number is simply $C(p, N-1)$, because constraining the hyperplanes to go through a particular point P makes the problem effectively $(N-1)$-dimensional; as illustrated in Fig. 5.12, we can project the whole problem onto an $(N-1)$-dimensional plane

[7]Nor is it well defined unless a distribution function is specified.

perpendicular to OP, since any displacement of a point along the OP direction cannot affect which side it is of any hyperplane containing OP.

We thereby obtain the **recursion relation**

$$C(p+1,N) = C(p,N) + C(p,N-1). \tag{5.65}$$

Iterating this equation for p, $p-1$, $p-2$, ..., 1 yields

$$C(p,N) = \binom{p-1}{0}C(1,N) + \binom{p-1}{1}C(1,N-1) + \ldots + \binom{p-1}{p-1}C(1,N-p+1). \tag{5.66}$$

For $p \leq N$ this is easy to handle, because $C(1,N) = 2$ for all N; one point can be colored red or black. For $p > N$ the second argument of C becomes 0 or negative in some terms, but these terms can be eliminated by taking $C(p,N) = 0$ for $N \leq 0$. It is easy to check that this choice is consistent with the recursion relation (5.65), and with $C(p,1) = 2$ (in one dimension the only "hyperplane" is a point at the origin, allowing two dichotomies). Thus (5.66) makes sense for all values of p and N and can be written as

$$C(p,N) = 2\sum_{i=0}^{N-1}\binom{p-1}{i} \tag{5.67}$$

if we use the standard convention that $\binom{n}{m} = 0$ for $m > n$. Equation (5.67) was used to plot Fig. 5.11, thus completing the demonstration.

It is actually easy to show from the symmetry $\binom{2n}{n-m} = \binom{2n}{n+m}$ of binomial coefficients that

$$C(2N,N) = 2^{p-1} \tag{5.68}$$

so the curve goes through 1/2 at $p = 2N$. To show analytically that the transition sharpens up for increasing N, one can appeal to the large N Gaussian limit of the binomial coefficients, which leads to

$$C(p,N)/2^p \approx \frac{1}{2}\left[1 + \operatorname{erf}\left(\sqrt{\frac{p}{2}}\left(\frac{2N}{p} - 1\right)\right)\right] \tag{5.69}$$

for large N.

It is worth noting that $C(p,N) = 2^p$ if $p \leq N$ (this is shown on page 155). So *any* coloring of up to N points is linearly separable, provided only that the points are in general position. For N or fewer points general position is equivalent to linear independence, so the sufficient conditions for a solution are exactly the same in the threshold and continuous-valued networks. But this is not true, of course, for $p > N$.