CHAPTER 5

Training Hidden Units: The Generalized Delta Rule

In this chapter, we introduce the back propagation learning procedure for learning internal representations. We begin by describing the history of the ideas and problems that make clear the need for back propagation. We then describe the procedure, focusing on the goal of helping the student gain a clear understanding of gradient descent learning and how it is used in training PDP networks. The exercises are constructed to allow the reader to explore the basic features of the back propagation paradigm. At the end of the chapter, there is a separate section on extensions of the basic paradigm, including three variants we call *cascaded* back propagation networks, *recurrent* networks, and *sequential* networks. Exercises are provided for each type of extension.

BACKGROUND

The pattern associator described in the previous chapter has been known since the late 1950s, when variants of what we have called the delta rule were first proposed. In one version, in which output units were linear threshold units, it was known as the perceptron (cf. Rosenblatt, 1959, 1962). In another version, in which the output units were purely linear, it was known as the LMS or least mean square associator (cf. Widrow & Hoff, 1960). Important theorems were proved about both of these versions. In the case of the perceptron, there was the so-called perceptron convergence theorem. In this theorem, the major paradigm is pattern classification. There is a set of binary input vectors, each of which can be said to belong to one of two classes. The system is to learn a set of connection strengths

and a threshold value so that it can correctly classify each of the input vectors. The basic structure of the perceptron is illustrated in Figure 1. The perceptron learning procedure is the following: An input vector is presented to the system (i.e., the input units are given an activation of 1 if the corresponding value of the input vector is 1 and are given 0 otherwise). The net input to the output unit is computed: $net = \sum_i w_i i_i$. If *net* is greater than the threshold θ, the unit is turned on, otherwise it is turned off. Then the response is compared with the actual category of the input vector. If the vector was correctly categorized, then no change is made to the weights. If, however, the output unit turns on when the input vector is in category 0, then the weights and thresholds are modified as follows: The threshold is incremented by 1 (to make it less likely that the output unit will come on if the same vector were presented again). If input i_i is 0, no change is made in the weight w_i (that weight could not have contributed to its having turned on). However, if $i_i = 1$, then w_i is decremented by 1. In this way, the system will not be as likely to turn on the next time this input vector is presented. On the other hand, if the output unit does not come on when it is supposed to, the opposite changes are made. That is, the threshold is decremented, and those weights connecting the output units to input units that are on are incremented.

Mathematically, this amounts to the following: The output, o, is given by

$$o = \begin{cases} 1 \text{ if } net = \sum_i w_i i_i > \theta \\ 0 \text{ otherwise.} \end{cases}$$

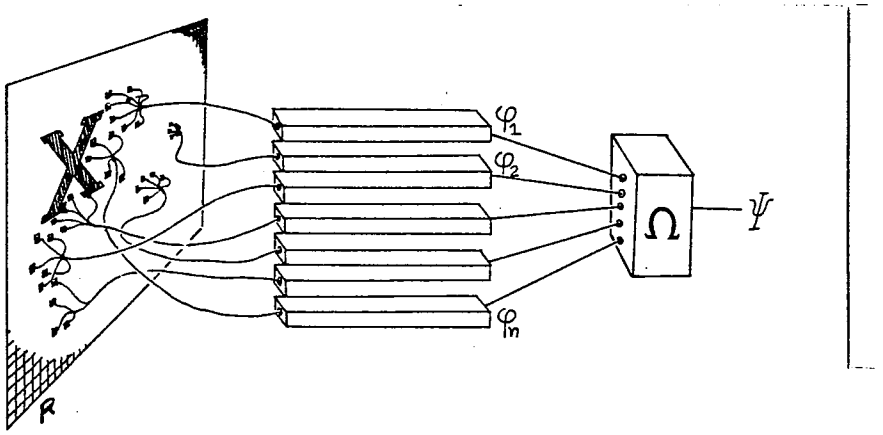

FIGURE 1. The one-layer perceptron analyzed by Minsky and Papert. (From *Perceptrons* by M. L. Minsky and S. Papert, 1969, Cambridge, MA: MIT Press. Copyright 1969 by MIT Press. Reprinted by permission.)

The change in the threshold, $\Delta\theta$, is given by

$$\Delta\theta = -(t_p - o_p) = -\delta_p$$

where p indexes the particular pattern being presented, t_p is the target value indicating the correct classification of that input pattern, and δ_p is the difference between the target and the actual output of the network. Finally, the changes in the weights, Δw_i, are given by

$$\Delta w_i = (t_p - o_p) i_{pi} = \delta_p i_{pi}.$$

The remarkable thing about this procedure is that, in spite of its simplicity, such a system is guaranteed to find a set of weights that correctly classifies the input vectors *if such a set of weights exists.* Moreover, since the learning procedure can be applied independently to each of a set of output units, the perceptron learning procedure will find the appropriate mapping from a set of input vectors onto a set of output vectors—*if such a mapping exists.* Unfortunately, as indicated in Chapter 4, such a mapping does not always exist, and this is the major problem for the perceptron learning procedure.

In their famous book *Perceptrons,* Minsky and Papert (1969) document the limitations of the perceptron. The simplest example of a function that cannot be computed by the perceptron is the exclusive-or (XOR), illustrated in Table 1. It should be clear enough why this problem is impossible. In order for a perceptron to solve this problem, the following four inequalities must be satisfied:

$$0 \times w_1 + 0 \times w_2 < \theta \Rightarrow 0 < \theta$$

$$0 \times w_1 + 1 \times w_2 > \theta \Rightarrow w_1 > \theta$$

$$1 \times w_1 + 0 \times w_2 > \theta \Rightarrow w_2 > \theta$$

$$1 \times w_1 + 1 \times w_2 < \theta \Rightarrow w_1 + w_2 < \theta$$

Obviously, we can't have both w_1 and w_2 greater than θ while their sum, $w_1 + w_2$, is less than θ. There is a simple geometric interpretation of the class of problems that can be solved by a perceptron: It is the class of

TABLE 1

Input Patterns		Output Patterns
00	→	0
01	→	1
10	→	1
11	→	0

(From *PDP:8*, p. 319)

linearly separable functions. This can easily be illustrated for two-dimensional problems such as XOR. Figure 2 shows a simple network with two inputs and a single output and illustrates three two-dimensional functions: the AND, the OR, and the XOR. The first two can be computed by the network; the third cannot. In these geometrical representations, the input patterns are represented as coordinates in space. In the case of a binary two-dimensional problem like XOR, these coordinates constitute the vertices of a square. The pattern 00 is represented at the lower left of the square, the pattern 10 as the lower right, and so on. The function to be computed is then represented by labeling each vertex with a 1 or 0 depending on which class the corresponding input pattern belongs to. The perceptron can solve any function in which a single line can be drawn through the space such that all of those labeled "0" on are one side of the line and those labeled "1" are on the other side. This can easily be done for AND and OR, but not for XOR. The line corresponds to the equation $i_1w_1+i_2w_2=\theta$. In three dimensions there is a plane, $i_1w_1+i_2w_2+i_3w_3=\theta$, that corresponds to the line. In higher dimensions there is a corresponding

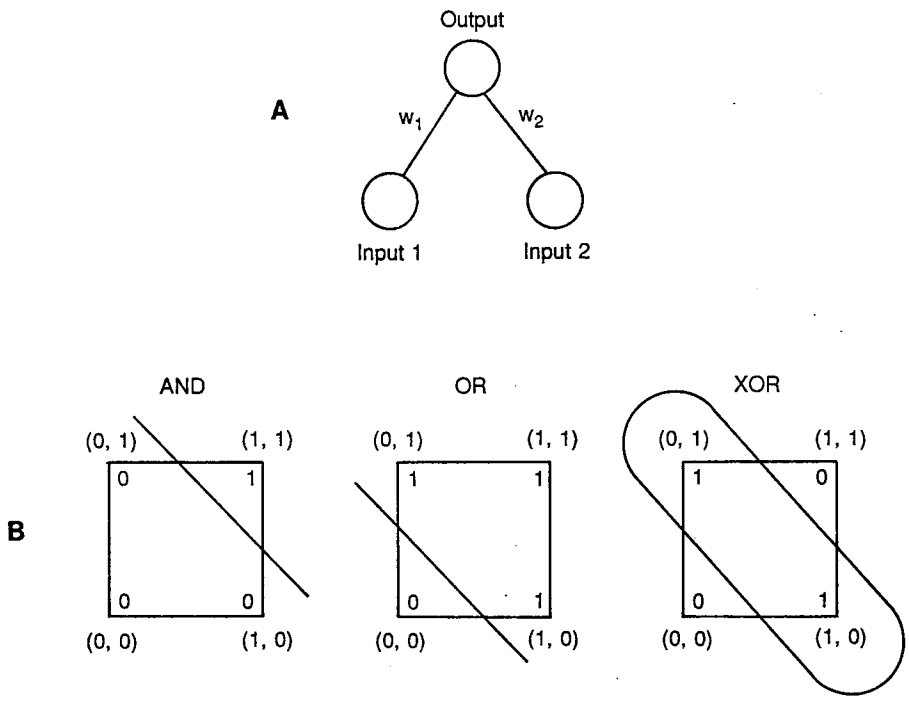

FIGURE 2. *A:* A simple network that can solve the two-dimensional AND and OR functions but cannot solve the XOR function. *B:* Geometric representations of the three problems. See text for details.

TABLE 2

Input Patterns		Output Patterns
000	→	0
010	→	1
100	→	1
111	→	0

(From *PDP:8*, p. 319.)

hyperplane, $\sum_{i=1}^{n} w_i i_i = \theta$. All functions for which there exists such a plane are called *linearly separable.* Now consider the function in Table 2 and illustrated in Figure 3. This is a three-dimensional problem in which the first two dimensions are identical to the XOR and the third dimension is the AND of the first two dimensions. (That is, the third dimension is 1 whenever both of the first two dimensions are 1, otherwise it is 0). Figure 3 shows how this problem can be represented in three dimensions. The figure also shows how the addition of the third dimension allows a plane to separate the patterns classified in category 0 from those in category 1. Thus, we see that the XOR is not solvable in two dimensions, but if we add the appropriate third dimension, that is, the appropriate *new feature,* the problem *is* solvable. Moreover, as indicated in Figure 4, if you allow a multilayered perceptron, it is possible to take the original two-dimensional problem and convert it into the appropriate three-dimensional problem so it can be solved. Indeed, as Minsky and Papert knew, it is always possible to convert any unsolvable problem into a solvable one in a multilayer perceptron. In the more general case of multilayer networks, we categorize units

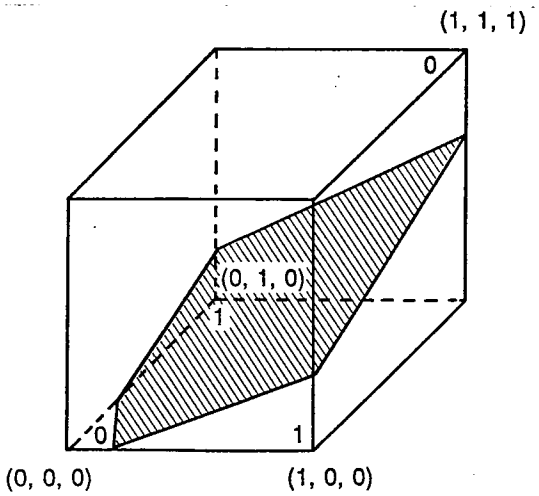

FIGURE 3. The three-dimensional solution of the XOR problem.

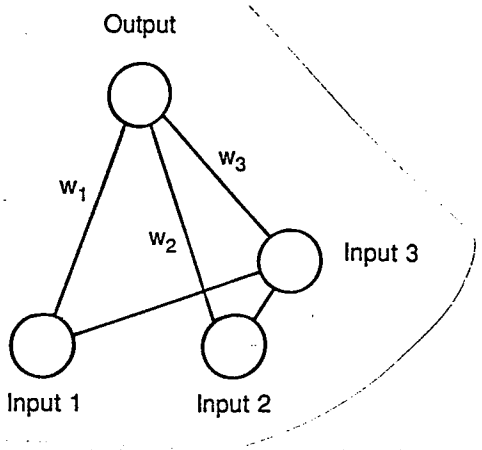

FIGURE 4. A multilayer network that converts the two-dimensional XOR problem into a three-dimensional linearly separable problem.

into three classes: *input units,* which receive the input patterns directly; *output units,* which have associated *teaching* or *target* inputs; and *hidden units,* which neither receive inputs directly nor are given direct feedback. This is the stock of units from which new features and new internal representations can be created. The problem is to know which new features are required to solve the problem at hand. In short, we must be able to learn intermediate layers. The question is, how? The original perceptron learning procedure does not apply to more than one layer. Minsky and Papert believed that no such general procedure could be found. To examine how such a procedure can be developed it is useful to consider the other major one-layer learning system of the 1950s and early 1960s, namely, the *least-mean-square (LMS)* learning procedure of Widrow and Hoff (1960).

Minimizing Mean Squared Error

The LMS procedure makes use of the delta rule for adjusting connection strengths; the perceptron convergence procedure is very similar, differing only in that linear threshold units are used instead of units with continuous-valued outputs. We use the term *LMS procedure* here to stress the fact that this family of learning rules may be viewed as minimizing a measure of the error in their performance.

The LMS procedure cannot be directly applied when the output units are linear threshold units (like the perceptron). It has been applied most often with purely linear output units. In this case the activation of an output

unit, o_i, is simply given by $o_i = \sum_j w_{ij} i_j$. The error function, as indicated by the name least-mean-square, is the summed squared error. That is, the total error, E, is defined to be

$$E = \sum_p E_p = \sum_p \sum_i (t_{pi} - o_{pi})^2 \tag{1}$$

where the index p ranges over the set of input patterns, i ranges over the set of output units, and E_p represents the error on pattern p. The variable t_{pi} is the desired output, or *target,* for the ith output unit when the pth pattern has been presented, and o_{pi} is the actual output of the ith output unit when pattern p has been presented. The object is to find a set of weights that minimizes this function. It is useful to consider how the error varies as a function of any given weight in the system. Figure 5 illustrates the nature of this dependence. In the case of the simple single-layered linear system, we always get a smooth error function such as the one shown in the figure. The LMS procedure finds the values of all of the weights that minimize this function using a method called *gradient descent.* That is, after each pattern has been presented, the error on that pattern is computed and each weight is moved "down" the error gradient toward its minimum value for that pattern. Since we cannot map out the entire error function on each pattern presentation, we must find a simple procedure for determining, for each weight, how much to increase or decrease each weight. The idea of gradient descent is to make a change in the weight proportional to the

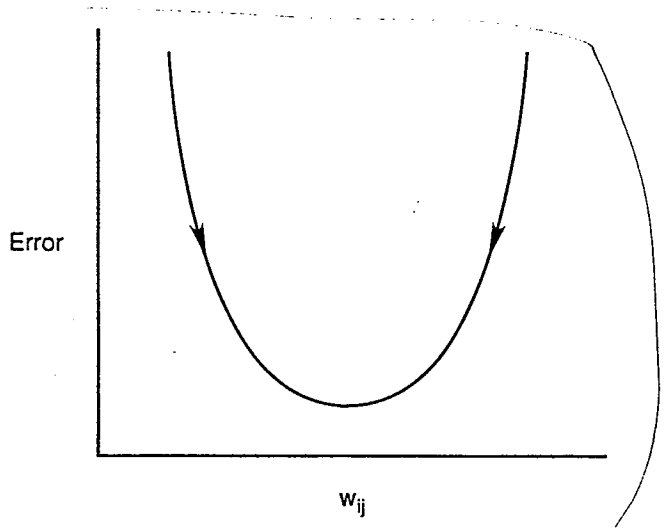

FIGURE 5. Typical curve showing the relationship between overall error and changes in a single weight in the network.

negative of the derivative of the error, as measured on the current pattern, with respect to each weight.[1] Thus the learning rule becomes

$$\Delta w_{ij} = -k\frac{\partial E_p}{\partial w_{ij}}$$

where k is the constant of proportionality. Interestingly, carrying out the derivative of the error measure in Equation 1 we get

$$\Delta w_{ij} = \epsilon \delta_{pi} i_{pj}$$

where $\epsilon = 2k$ and $\delta_{pi} = t_{pi} - o_{pi}$ is the difference between the target for unit i on pattern p and the actual output produced by the network. This is exactly the delta learning rule described in Equation 15 from Chapter 4. It should also be noted that this rule is essentially the same as that for the perceptron. In the perceptron the learning rate was 1 (i.e., we made unit changes in the weights) and the units were binary, but the rule itself is the same: the weights are changed proportionally to the difference between target and output times the input.

If we change each weight according to this rule, each weight is moved toward its own minimum and we think of the system as moving downhill in *weight-space* until it reaches its minimum error value. When all of the weights have reached their minimum points, the system has reached equilibrium. If the system is able to solve the problem entirely, the system will reach zero error and the weights will no longer be modified. On the other hand, if the network is unable to get the problem exactly right, it will find a set of weights that produces as small an error as possible.

In order to get a fuller understanding of this process it is useful to carefully consider the entire error space rather than a one-dimensional slice. In general this is very difficult to do because of the difficulty of depicting and visualizing high-dimensional spaces. However, we can usefully go from one to two dimensions by considering a network with exactly two weights. Consider, as an example, a linear network with two input units and one output unit with the task of finding a set of weights that comes as close as possible to performing the function OR. Assume the network has just two weights and no bias terms like the network in Figure 2A. We can then give some idea of the shape of the space by making a contour map of the error surface.

Figure 6 shows the contour map. In this case the space is shaped like a kind of oblong bowl. It is relatively flat on the bottom and rises sharply on the sides. Each equal error contour is elliptically shaped. The arrows

[1] It should be clear from Figure 5 why we want the negation of the derivative. If the weight is above the minimum value, the slope at that point is *positive* and we want to *decrease* the weight; thus when the slope is positive we add a negative amount to the weight. On the other hand, if the weight is too small, the error curve has a negative slope at that point, so we want to add a positive amount to the weight.

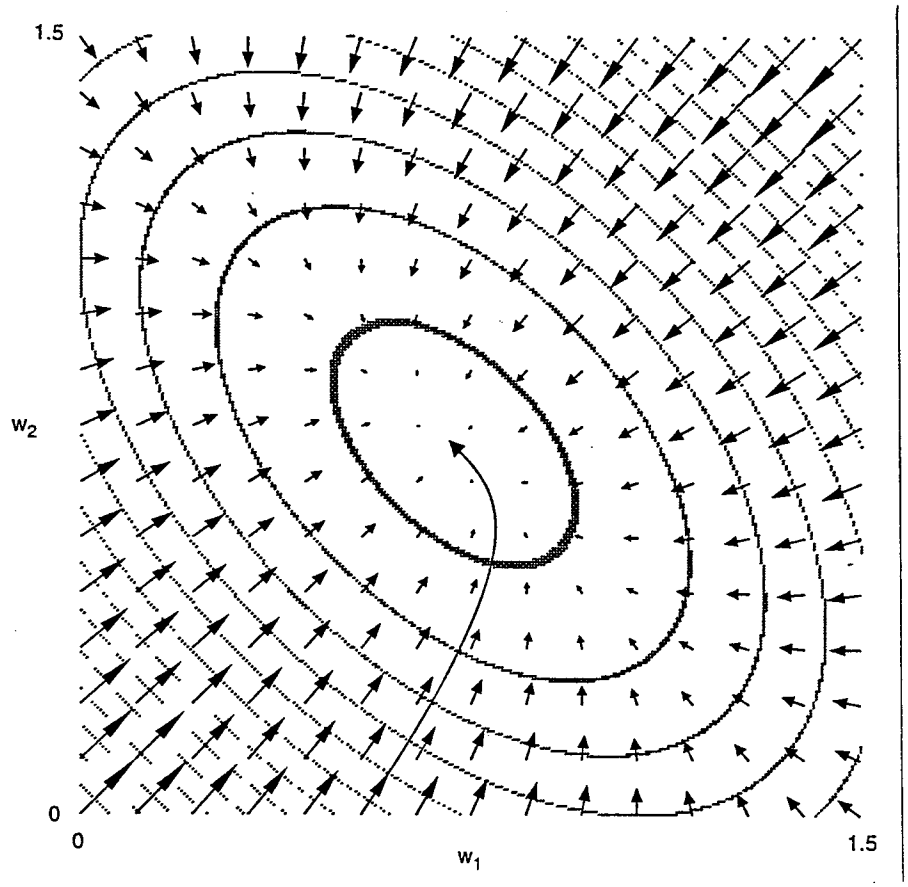

FIGURE 6. A contour map illustrating the error surface with respect to the two weights w_1 and w_2, for the OR problem in a linear network with two weights and no bias term. Note that the OR problem cannot be solved perfectly in a linear system. The minimum sum squared error over the four input-output pairs occurs when $w_1 = w_2 = 0.75$. (The input-output pairs are $00 \rightarrow 0$, $01 \rightarrow 1$, $10 \rightarrow 1$, and $11 \rightarrow 1$.)

around the ellipses represent the derivatives of the two weights at those points and thus represent the directions and magnitudes of weight changes at each point on the error surface. The changes are relatively large where the sides of the bowl are relatively steep and become smaller and smaller as we move into the central minimum. The long, curved arrow represents a typical trajectory in weight-space from a starting point far from the minimum down to the actual minimum in the space. The weights trace a curved trajectory following the arrows and crossing the contour lines at right angles.

The figure illustrates an important aspect of gradient descent learning. This is the fact that gradient descent involves making larger changes to parameters that will have the biggest effect on the measure being

minimized. In this case, the LMS procedure makes changes to the weights proportional to the effect they will have on the summed squared error. The resulting total change to the weights is a vector that points in the direction in which the error drops most steeply.

The Back Propagation Rule

Although this simple linear pattern associator is a useful model for understanding the dynamics of gradient descent learning, it is not useful for solving problems such as the XOR problem mentioned above. As pointed out in *PDP:2*, linear systems cannot compute more in multiple layers than they can in a single layer. The basic idea of the back propagation method of learning is to combine a nonlinear perceptron-like system capable of making decisions with the objective error function of LMS and gradient descent. To do this, we must be able to readily compute the derivative of the error function with respect to *any weight in the network* and then change that weight according to the rule

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

We will not bother with the mathematics here, since it is presented in detail in *PDP:8*. Suffice it to say, that with an appropriate choice of nonlinear function we can perform the differentiation and derive the back propagation learning rule. The rule has exactly the same form as the learning rules described above, namely,[2]

$$\Delta w_{ij} = \epsilon \delta_{pi} a_{pj}.$$

The weight on each line should be changed by an amount proportional to the product of a term called δ, available to the unit receiving input along that line, times the activation, a, of the unit sending activation along that line.[3] The difference is in the exact determination of the δ term. Essentially, δ_{pi} represents the effect of a change in the net input to unit j on the output of unit i in pattern p. The determination of δ is a recursive process that starts with the output units. If a unit is an output unit, its δ is very similar to the one used in the standard delta rule. It is given by

$$\delta_{pi} = (t_{pi} - a_{pi}) f'_i(net_{pi})$$

---

[2] Note that the symbol η was used for the learning rate parameter in *PDP:8*. We use ϵ here for consistency with other chapters in this volume.

[3] In the networks we will be considering in this chapter, the output of a unit is equal to its activation. We use the symbol a to designate this variable. This symbol can be used for any unit, be it an input unit, an output unit, or a hidden unit.

where $net_{pi} = \sum_j w_{ij} a_{pj} + bias_i$ and $f'_i(net_{pi})$ is the derivative of the activation function with respect to a change in the net input to the unit. Note that $bias_i$ is a bias that has a similar function to the threshold, θ, in the perceptron.[4]

The δ term for hidden units for which there is no specified target is determined recursively in terms of the δ terms of the units to which it directly connects and the weights of those connections. That is,

$$\delta_{pi} = f'_i(net_{pi}) \sum_k \delta_{pk} w_{ki}$$

whenever the unit is not an output unit.

The application of the back propagation rule, then, involves two phases: During the first phase the input is presented and propagated forward through the network to compute the output value a_{pj} for each unit. This output is then compared with the target, resulting in a δ term for each output unit. The second phase involves a backward pass through the network (analogous to the initial forward pass) during which the δ term is computed for each unit in the network. This second, backward pass allows the recursive computation of δ as indicated above. Once these two phases are complete, we can compute, for each weight, the product of the δ term associated with the unit it projects to times the activation of the unit it projects from. Henceforth we will call this product the *weight error derivative* since it is proportional to (minus) the derivative of the error with respect to the weight. As will be discussed later, these weight error derivatives can then be used to compute actual weight changes on a pattern-by-pattern basis, or they may be accumulated over the ensemble of patterns.

The activation function. The derivation of the back propagation learning rule requires that the derivative of the activation function, $f'_i(net_i)$, exists. It is interesting to note that the linear threshold function, on which the perceptron is based, is discontinuous and hence will not suffice for back propagation. Similarly, since a linear system achieves no advantage from hidden units, a linear activation function will not suffice either. Thus, we need a continuous, nonlinear activation function. In most of our work on back propagation and in the program presented in this chapter, we have used the *logistic* activation function in which

$$a_{pi} = \frac{1}{1 + e^{-net_{pi}}}.$$

[4] Note that the values of the bias can be learned just like any other weights. We simply imagine that the bias is the weight from a unit that is always on.

In order to apply our learning rule, we need to know the derivative of this function with respect to its total input, net_{pi}. It is easy to show that this derivative is given by

$$\frac{da_{pi}}{dnet_{pi}} = a_{pi}(1 - a_{pi}).$$

Thus, for the logistic activation function, the error signal, δ_{pi}, for an output unit is given by

$$\delta_{pi} = (t_{pi} - a_{pi})a_{pi}(1 - a_{pi}),$$

and the error for an arbitrary hidden u_i is given by

$$\delta_{pi} = a_{pi}(1 - a_{pi})\sum_k \delta_{pk} w_{jk}.$$

It should be noted that the derivative, $a_{pi}(1 - a_{pi})$, reaches its maximum at $a_{pi} = 0.5$ and, since $0 \leqslant a_{pi} \leqslant 1$, approaches its minimum as a_{pi} approaches 0 or 1. Since the amount of change in a given weight is proportional to this derivative, weights will be changed most for those units that are near their midrange and, in some sense, not yet committed to being either on or off. This feature, we believe, contributes to the stability of the learning of the system.

Local minima. Like the simpler LMS learning paradigm, back propagation is a gradient descent procedure. Essentially, the system will follow the contour of the error surface—always moving downhill in the direction of steepest descent. This is no particular problem for the single-layer linear model. These systems always have bowl-shaped error surfaces. However, in multilayer networks there is the possibility of rather more complex surfaces with many minima. Some of the minima constitute solutions to the problems in which the system reaches an errorless state. All such minima are *global* minima. However, it is possible for some of the minima to be deeper than others. In this case, a gradient descent method may not find the *best* possible solution to the problem at hand. Part of the study of back propagation networks and learning involves a study of how frequently and under what conditions local minima occur. In problems with many hidden units, local minima seem quite rare. However with few hidden units, local minima can be more common. Figure 7 shows a very simple network in which we can demonstrate these phenomena. The network involves a single input unit, a single hidden unit, and a single output unit (a 1:1:1 network, for short). The problem is to copy the value of the input unit to the output unit. There are two basic ways in which the network can solve the problem. It can have positive biases on the hidden unit and on the output unit and large negative connections from the input unit to the hidden unit and from the hidden unit to the output unit, or it can have large negative biases on the two units and large positive weights from the input unit to the

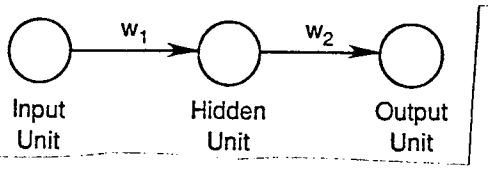

FIGURE 7. A 1:1:1 network, consisting of one input unit, one hidden unit, and one output unit.

hidden unit and from the hidden unit to the output unit. These solutions are illustrated in Table 3. In the first case, the solution works as follows: Imagine first that the input unit takes on a value of 0. In this case, there will be no activation from the input unit to the hidden unit, but the bias on the hidden unit will turn it on. Then the hidden unit has a *strong negative* connection to the output unit so it will be turned off, as required in this case. Now suppose that the input unit is set to 1. In this case, the strong inhibitory connection from the input to the hidden unit will turn the hidden unit off. Thus, no activation will flow from the hidden unit to the output unit. In this case, the positive bias on the output unit will turn it on and the problem will be solved. Now consider the second class of solutions. For this case, the connections among units are positive and the biases are negative. When the input unit is off, it cannot turn on the hidden unit. Since the hidden unit has a negative bias, it too will be off. The output unit, then, will not receive any input from the hidden unit and since its bias is negative, it too will turn off as required for zero input. Finally, if the input unit is turned on, the strong positive connection from the input unit to the hidden unit will turn on the hidden unit. This in turn will turn on the output unit as required. Thus we have, it appears, two symmetric solutions to the problem. Depending on the random starting state, the system will end up in one or the other of these *global* minima.

Interestingly, it is a simple matter to convert this problem to one with one local and one global minimum simply by setting the biases to 0 and not allowing them to change. In this case, the minima correspond to roughly the same two solutions as before. In one case, which is the global minimum as it turns out, both connections are large and negative. These minima are also illustrated in Table 3. Consider first what happens with

TABLE 3

WEIGHTS AND BIASES OF THE SOLUTIONS FOR A 1:1:1 NETWORK

Minima	w_1	w_2	$bias_1$	$bias_2$
Global	−8	−8	+4	+4
Global	+8	+8	−4	−4
Global	−8	−8	0	0
Local	+8	+0.73	0	0

both weights negative. When the input unit is turned off, the hidden unit receives no input. Since the bias is 0, the hidden unit has a net input of 0. A net input of 0 causes the hidden unit to take on a value of 0.5. The 0.5 input from the hidden unit, coupled with a large negative connection from the hidden unit to the output unit, is sufficient to turn off the output unit as required. On the other hand, when the input unit is turned on, it turns off the hidden unit. When the hidden unit is off, the output unit receives a net input of 0 and takes on a value of 0.5 rather than the desired value of 1.0. Thus there is an error of 0.5 and a squared error of 0.25. This, it turns out, is the best the system can do with zero biases. Now consider what happens if both connections are positive. When the input unit is off, the hidden unit takes on a value of 0.5. Since the output is intended to be 0 in this case, there is pressure for the weight from the hidden unit to the output unit to be small. On the other hand, when the input unit is on, it turns on the hidden unit. Since the output unit is to be on in this case, there is pressure for the weight to be large so it can turn on the output unit. In fact, these two pressures balance off and the system finds a compromise value of about 0.73. This compromise yields a summed squared error of about 0.45—a local minima.

Usually, it is difficult to see why a network has been caught in a local minimum. However, in this very simple case, we have only two weights and can produce a contour map for the error space. The map is shown in Figure 8. It is perhaps difficult to visualize, but the map roughly shows a saddle shape. It is high on the upper left and lower right and slopes down toward the center. It then slopes off on each side toward the two minima. If the initial values of the weights begin below the antidiagonal (that is, below the line $w_1 + w_2 = 0$), the system will follow the contours down and to the left into the minimum in which both weights are negative. If, however, the system begins above the antidiagonal, the system will follow the slope into the upper right quadrant in which both weights are positive. Eventually, the system moves into a gently sloping valley in which the weight from the hidden unit to the output unit is almost constant at about 0.73 and the weight from the input unit to the hidden unit is slowly increasing. It is slowly being sucked into a local minimum. The directed arrows superimposed on the map illustrate the lines of force and illustrate these dynamics. The long arrows represent two trajectories through weight-space for two different starting points.

It is rare that we can create such a simple illustration of the dynamics of weight-spaces and how local minima come about. However, it is likely that many of our spaces contain these kinds of saddle-shaped error surfaces. Sometimes, as when the biases are free to move, there is a global minimum on either side of the saddle point. In this case, it doesn't matter which way you move off. At other times, such as in Figure 8, the two sides are of different depths. There is no way the system can sense the depth of a minimum from the edge, and once it has slipped in there is no way out. Importantly, however, we find that high-dimensional spaces (with many

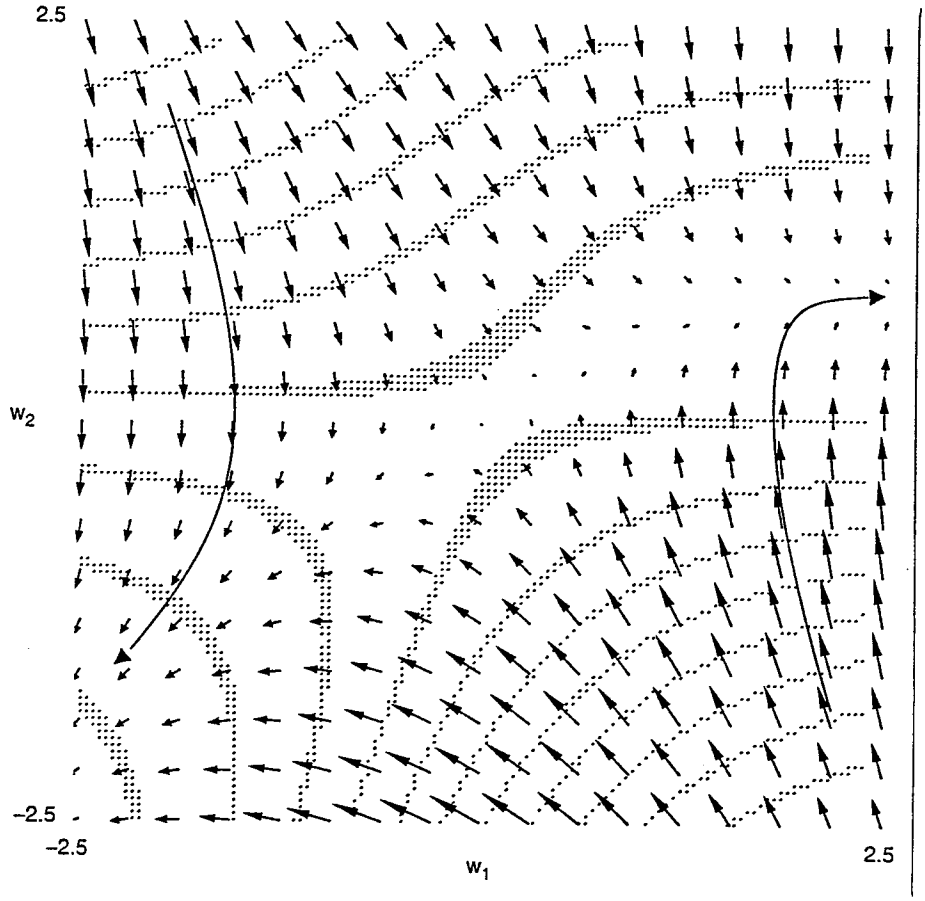

FIGURE 8. A contour map for the 1:1:1 identity problem with biases fixed at 0. The map show a local minimum in the positive quadrant and a global minimum in the lower left-hand negative quadrant. Overall the error surface is saddle-shaped. See the text for further explanation.

weights) have relatively few local minima. It seems that the system can always, as it were, slip along another dimension to find a path out of most local minima.

Momentum. Our learning procedure requires only that the change in weight be proportional to the weight error derivative. True gradient descent requires that infinitesimal steps be taken. The constant of proportionality, ϵ, is the learning rate in our procedure. The larger this constant, the larger the changes in the weights. For practical purposes we choose a learning rate that is as large as possible without leading to oscillation. This offers the most rapid learning. One way to increase the learning rate without leading to oscillation is to modify the back propagation learning

rule to include a *momentum* term. This can be accomplished by the following rule:

$$\Delta w_{ij}(n+1) = \epsilon(\delta_{pi}a_{pj}) + \alpha\Delta w_{ij}(n)$$

where the subscript n indexes the presentation number and α is a constant that determines the effect of past weight changes on the current direction of movement in weight space. This provides a kind of momentum in weight-space that effectively filters out high-frequency variations of the error surface in the weight-space. This is useful in spaces containing long ravines that are characterized by sharp curvature across the ravine and a gently sloping floor. The sharp curvature tends to cause divergent oscillations across the ravine. To prevent these it is necessary to take very small steps, but this causes very slow progress along the ravine. The momentum filters out the high curvature and thus allows the effective weight steps to be bigger. In most of the simulations reported in *PDP:8*, α was about 0.9. Our experience has been that we get the same solutions by setting $\alpha = 0$ and reducing the size of ϵ, but the system learns much faster overall with larger values of α and ϵ.

Symmetry breaking. Our learning procedure has one more problem that can be readily overcome and this is the problem of symmetry breaking. If all weights start out with equal values and if the solution requires that unequal weights be developed, the system can never learn. This is because error is propagated back through the weights in proportion to the values of the weights. This means that all hidden units connected directly to the output units will get identical error signals, and, since the weight changes depend on the error signals, the weights from those units to the output units must always be the same. The system is starting out at a kind of unstable equilibrium point that keeps the weights equal, but it is higher than some neighboring points on the error surface, and once it moves away to one of these points, it will never return. We counteract this problem by starting the system with small random weights. Under these conditions symmetry problems of this kind do not arise. This can be seen in Figure 8. If the system starts at exactly (0,0), there is no pressure for it to move at all and the system will not learn, but if it starts anywhere off of the antidiagonal, it will eventually end up in one minimum or the other.

Learning by pattern or by epoch. The derivation of the back propagation paradigm supposes that we are taking the derivative of the error function summed over all patterns. In this case, we might imagine that we would present all patterns and then sum the derivatives before changing the weights. Instead, we can compute the derivatives on each pattern and make the changes to the weights after each pattern rather than after each epoch. If the learning rate is small, there is little difference between the two procedures, and the version in which weights are changed after each pattern