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

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

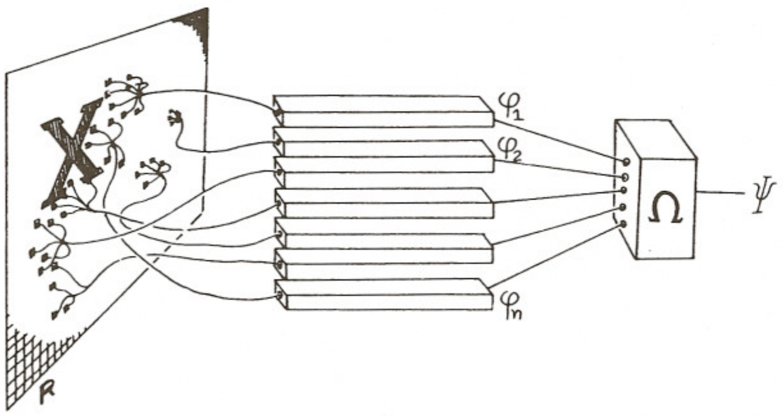


FIGURE 1. The one-layer perceptron analyzed by Minsky and Papert. (From *Perceptrons* by M. L. Minsky and S. Papert, 1969, Cambridge, MA: MIT Press. Copyright 1969 by MIT Press. Reprinted by permission.)

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:

$$0 \times w_1 + 0 \times w_2 < \theta => 0 < \theta$$

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

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

$$1 \times w_1 + 1 \times w_2 < \theta => w_1 + w_2 < \theta$$

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

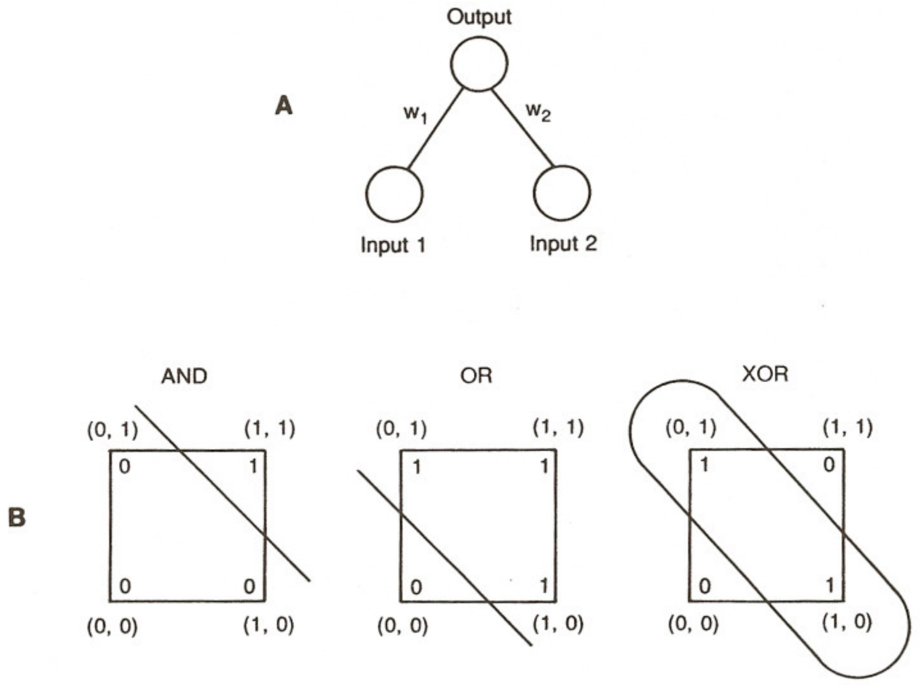


FIGURE 2. *A:* A simple network that can solve the two-dimensional AND and OR functions but cannot solve the XOR function. *B:* Geometric representations of the three problems. See text for details.

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

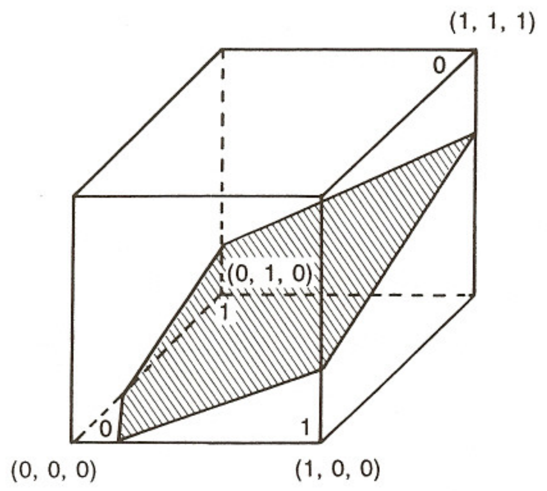


FIGURE 3. The three-dimensional solution of the XOR problem.

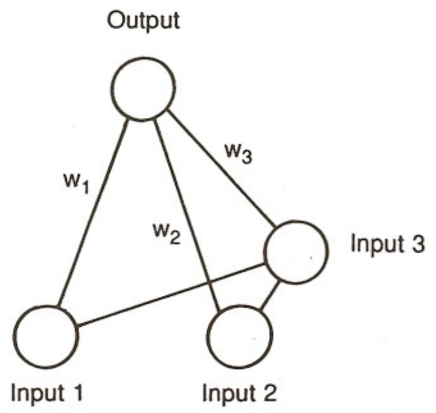


FIGURE 4. A multilayer network that converts the two-dimensional XOR problem into a three-dimensional linearly separable problem.

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

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

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

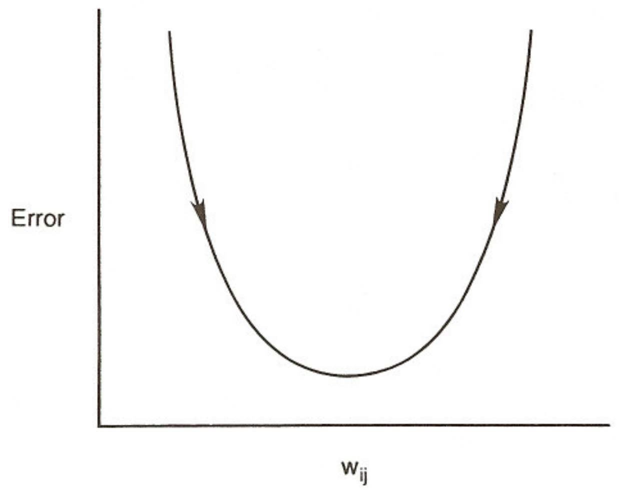


FIGURE 5. Typical curve showing the relationship between overall error and changes in a single weight in the network.

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.

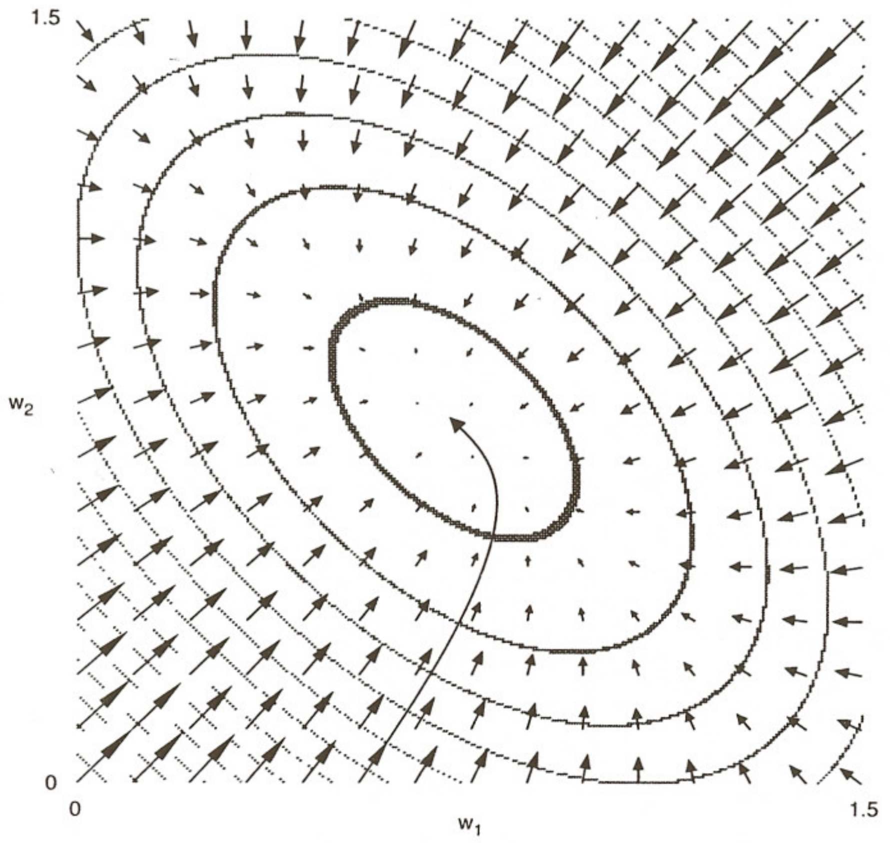


FIGURE 6. A contour map illustrating the error surface with respect to the two weights w_1 and w_2, for the OR problem in a linear network with two weights and no bias term. Note that the OR problem cannot be solved perfectly in a linear system. The minimum sum squared error over the four input-output 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 non-linear 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

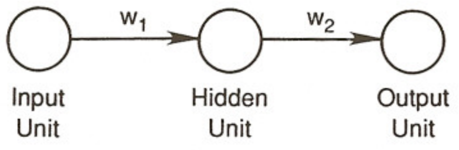


FIGURE 7. A 1:1:1 network, consisting of one input unit, one hidden unit, and one output unit.

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

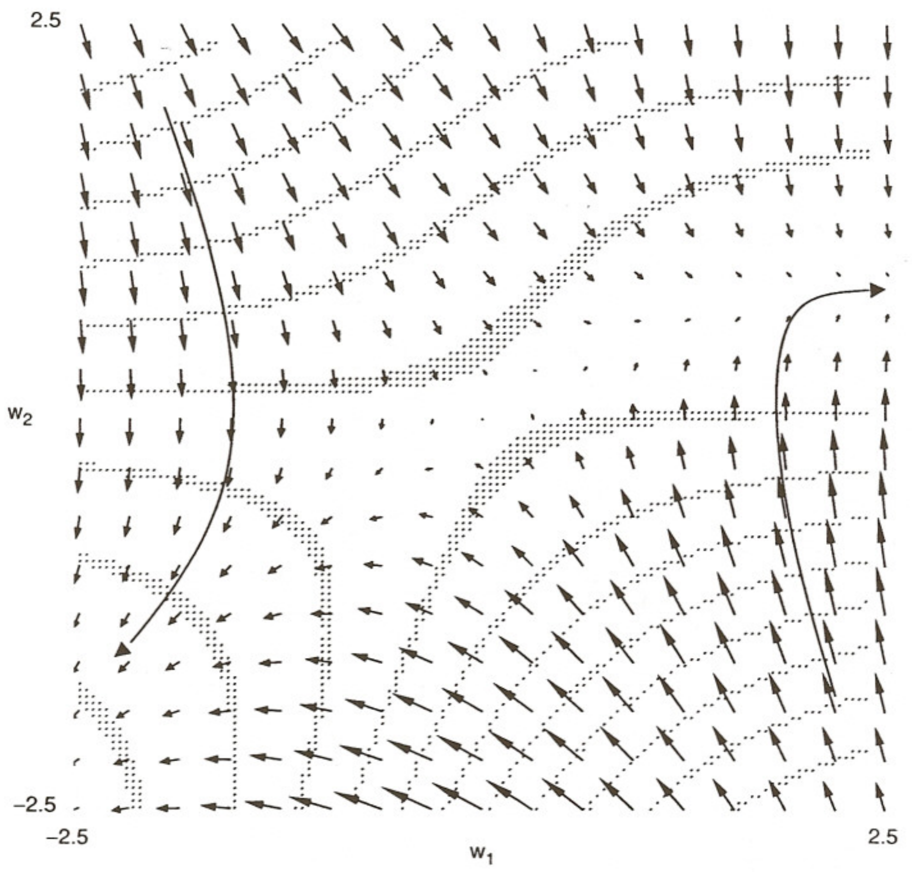


FIGURE 8. A contour map for the 1:1:1 identity problem with biases fixed at 0. The map show a local minimum in the positive quadrant and a global minimum in the lower left-hand negative quadrant. Overall the error surface is saddle-shaped. See the text for further explanation.

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 seems more satisfying—since there might be a very large set of patterns. The **bp** program introduced in the next section allows both of these options.

IMPLEMENTATION

The **bp** program implements the back propagation process just described. The program makes use of a network specification file, which indicates the architecture of the network. Networks are assumed to be feedforward only, with no recurrence, although limited recurrence can be simulated by linking weights to each other or by copying activations of output units back into the input units. These variations are described at the end of this chapter.

The network specifications indicate how many total units are in the network and, of these, how many are input units and how many are output units. The units in the network specification are assumed to be ordered in such a way that no unit ever receives input from a unit occurring later in the ordering. Thus, the first listed units are the input units, the next listed are the hidden units, and the last are the output units. Among the hidden units, if there are multiple layers, the units in the layer closest to the input are listed first, and so on.

Weights may constrained to be positive, negative, or linked. When a weight is "linked" that means that it is forced to have the same value as all of the other weights that it is linked to; there may be any number of "link groups" of weights. These constraints are imposed each time the weights are incremented during processing. Two other constraints are imposed only when weights are initialized; these constraints are either a fixed (floating-point) value to which the weight is initialized or a random value. For weights that are random, if they are constrained to be positive, they are initialized to a value between 0 and the value of a parameter called *wrange*; if the weights are constrained to be negative, the initialization value is between $-wrange$ and 0; otherwise, the initialization value is between $wrange/2$ and $-wrange/2$. Weights that are constrained to a fixed value are initialized to that value. Bias terms are treated like weights and may be subject to all the same kinds of constraints.

The program also allows the user to set individually for each weight and bias term whether the weight or bias will be modifiable. The weight modifiability information is recorded in a matrix called *epsilon*; there is also a corresponding vector called *bepsilon* that records the modifiability of the bias terms of each unit. When the network is initialized, *epsilon* and *bepsilon* are set to *lrate* (the value of the global learning rate parameter) or 0, depending on whether the weight is modifiable or not. If *lrate* is changed, all nonzero *epsilon* and *bepsilon* terms are set to the new value of *lrate*. (The conventions for setting up a network specification file are described in Appendix C.)

The **bp** program also makes use of a list of pattern pairs, each pair consisting of a name, an input pattern, and a target pattern.

Processing of a single pattern occurs as follows: A pattern pair is chosen, and the pattern of activation specified by the input pattern is clamped on the input units; that is, their activations are set to 1 or 0 based on the values found in the input pattern.

Next, activations are computed. For each noninput unit, the net input to the unit is computed and then the activation of the unit is set. This occurs in the order that the units are specified in the network specification, so that by the time each unit is encountered, the activations of all of the units that feed into it have already been set. The routine that performs this computation is

```
compute_output() {

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

In this code, note that *ninputs*, which is the number of input units, is also the index of the first hidden unit. The arrays *first_weight_to* and *last_weight_to* indicate which unit is the first and which is the last to project to each unit.[5]

Next, *error* and *delta* terms are computed. The *error* for a unit is equivalent to the partial derivative of the error with respect to a change in the *activation* of the unit. The *delta* for the unit is the partial derivative of the error with respect to a change in the *net input* to the unit.

First, the *delta* and *error* terms for all units are set to 0. Then, *error* terms are calculated for each output unit. For these units, *error* is the difference between the target and the obtained activation of the unit.

After the error has been computed for each output unit, we get to the "heart" of back propagation: the recursive computation of *error* and *delta* terms for hidden units. The program iterates backward over the units, starting with the last output unit. The first thing it does in each pass through the loop is set *delta* for the current unit, which is equal to the *error* for the unit times the derivative of the activation function (i.e., the activation of the unit times one minus its activation). Then, once it has *delta* for the current unit, the program passes this back to all units that have

[5] Note that for efficiency and other reasons, the weight indexes are not actually implemented in the form shown here. A description of the actual treatment of weight arrays is given in Appendix F.

connections coming into the current unit; this is the actual back propagation process. By the time a particular unit becomes the current unit, all of the units that it projects to will have already been processed, and all of its error will have been accumulated, so it is ready to have its *delta* computed. The code for this is as follows:

```
compute_error() {

  for (i = ninputs; i < nunits; i++) {
    error[i] = 0.0;
  }

  for (i = nunits-noutputs, t=0; i<nunits; t++, i++) {
    error[i] = target[t] - activation[i];
  }

  for (i = nunits - 1; i >= ninputs; i--) {
    delta[i] = error[i]*activation[i]*(1.0-activation[i]);
    for (j=first_weight_to[i]; j<last_weight_to[i]; j++)
      error[j] += delta[i] * weight[i][j];
  }
}
```

After computing error, if the *lflag* is nonzero, the weight error derivatives are then computed from the *delta*s and *activation*s. The error derivatives for the *bias* terms are also computed. (Recall that the *bias* terms are equivalent to weights to a unit from a unit whose activation is always 1.0.) These computations occur in the following routine:

```
compute_wed() {

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

Note that this routine adds the weight error derivatives occasioned by the present pattern into an array where they can potentially be accumulated over patterns.

Weight error derivatives actually lead to changes in the weights either after processing each pattern or after each entire epoch of processing. In either case, the computation that is actually performed needs to be clearly understood. For each weight, a delta weight is first calculated. The delta weight is equal to the accumulated weight error derivative plus a fraction of the previous delta weight, where the size of the fraction is determined by the parameter *momentum*. Then, this delta weight is added into the weight, so that the weight's new value is equal to its old value plus the delta weight. Again, the same computation is performed for all of the *bias* terms. The following routine performs these computations:

```
change_weights() {

  sum_linked_weds();

  for (i = ninputs; i < nunits; i++) {
    for (j=first_weight_to[i]; j<last_weight_to[i]; j++) {
      dweight[i][j] = epsilon[i][j]*wed[i][j] +
                        momentum*dweight[i][j];
      weight[i][j] += dweight[i][j];
      wed[i][j] = 0.0;
    }
    dbias[i] = bepsilon[i]*bed[i] + momentum*dbias[i];
    bias[i] += dbias[i];
    bed[i] = 0.0;
  }

  constrain_neg_pos();
}
```

Note that before the weights are actually changed, the *sum_linked_weds* routine is called. This routine adds together all the weight error derivative terms associated with all the weights that are linked together in the same link group. The idea of linking weights is to assure that all weights that are linked together always have the same value since conceptually they are thought of as being a single weight. Also, after the weights are changed, the *constrain_neg_pos* routine is called to make sure that the values assigned to the weights conform to the positive or negative constraints that have been imposed on them in the network specification file. Weights that are constrained to be positive are reset to 0 by *constrain_neg_pos* if *change_weights* trys to put them below 0, and weights that are constrained to be negative are reset to 0 if *change_weights* tries to put them above 0.

We have just described the processing activity that takes place for each input-target pair in each learning *trial.* Generally, learning is accomplished through a sequence of *epochs*, in which all pattern pairs are presented for one trial each during each epoch. The *change_weights* routine may be called once per pattern or only once per epoch. The presentation is either in sequential or permuted order. It is also possible to test the processing of patterns, either individually or by sequentially cycling through the whole list, with learning turned off. In this case, *compute_output* and *compute_error* are called, but *compute_wed* and *change_weights* are not called.

Whether or not learning is occurring, the program also computes summary statistics after processing each pattern. First it computes the pattern sum of squares (*pss*), equal to the squared error terms summed over all of the output units. Then it adds the *pss* to the total sum of squares (*tss*), which is just the cumulative sum of the *pss* for all patterns thus far processed within the current epoch.

Learning is carried out by the *strain* and *ptrain* commands. The first carries out training in sequential order, the second in permuted order. Training goes on for *nepochs* or until the value of *tss* becomes less than the value of a control parameter called *ecrit* for "error criterion." (Note that *strain* and *ptrain* do not check the *tss* until the end of each epoch, so they will always run through at least one epoch before returning.)

Modes and Measures

The basic back propagation procedure can be operated in either of two *learning grain* modes: *pattern* or *epoch*. If the learning grain is set to *pattern*, the weight error derivatives are computed for each pattern, and then *change_weights* is called, updating the delta weights and adding them to the weights before the next pattern is processed. If learning grain is set to *epoch*, the weight error derivatives are still computed for each pattern, but they are accumulated until the end of the epoch. In this case, then, delta weights are computed and added into the actual weight array only once per epoch.

In the **bp** program the principle measure of performance is the pattern sum of squares (*pss*) and the total sum of squares (*tss*). The user may optionally also compute an additional measure, the vector correlation of the present weight error derivatives with the previous weight error derivatives. The set of weight error derivatives can be thought of as a vector pointing in the steepest direction downhill in weight space; that is, it points down the error gradient in weight space. Thus, the vector correlation of these derivatives across successive epochs indicates whether the gradient is staying relatively stable or shifting from epoch to epoch. For example, a negative value of this correlation measure (called *gcor* for *gradient correlation*) indicates that the gradient is changing in direction. Since the *gcor* can be thought of as following the gradient, the mode switch for turning on this computation is called *follow*.

The only other mode available in the **bp** program is *cascade* mode. This mode allows activation to build up gradually rather than being computed in a single step as is usually the case in **bp**. A discussion of the implementation and use of this mode is provided later in the "Extensions" section.

RUNNING THE PROGRAM

The **bp** program is used much like earlier programs in this series, particularly **pa**. Like the other programs, it has a flexible architecture that must be specified using a *.net* file. The conventions used in these files are described in Appendix C. The program also makes use of a *.pat* file, in which the pairs of patterns to be used in training and testing the network are listed.

When networks are initialized, the weights are generally assigned according to a pseudo-random number generator. As in **pa** and **iac**, the *reset* command allows the user to repeat a simulation run with the same initial configuration used just before. (Another procedure for repeating the previous run is described in Ex. 5.1.) The *newstart* command generates a new random seed and seeds the random number generator with it.

Commands

All of the commands in **bp** will be familiar from previous programs. The only ones that are notably changed in usage from **pa** are the *test* and *get/ patterns* commands. In **bp**, *test* simply prompts for the name or number of a pattern from the pattern list and does not allow the option of specifying an arbitrary input pattern; *get/ patterns* works as before except that negative entries in the *.pat* file are treated specially in **bp**; see the *env/ ipattern* and *env/ tpattern* entries in the list of variables below.

Variables

There are several new variables and a few minor changes to the meaning of some familiar variables. These new and changed variables are described in the following list. As in previous programs, all of the variables in **bp** are accessed via the *set/* and *exam/* commands.

ncycles

The number of processing cycles run when the *cycle* routine is called. This program control variable is used in *cascade* mode, described later in this chapter.

stepsize

Size of the processing step taken before updating the screen and prompting for *return* in single step mode. Possible values are *cycle*, *ncycles*, *pattern*, *epoch*, and *nepochs*. When *cascade* mode is not on and if the value of *stepsize* is *cycle* or *ncycles*, the screen is updated after the forward processing sweep, then again after the backward processing sweep and any weight adjustments. When *cascade* mode is on and if the *stepsize* is *cycle*, updating occurs after each processing cycle; if *stepsize* is *ncycles*, updating occurs after *ncycles* of processing.

config/ bepsilon

For each bias term, there is an associated *bepsilon*, or modifiability parameter, just as for each weight. These are just like *epsilons* (see below), except that the user must only indicate one index since there is just one per unit.

config/ epsilon

For each weight, there is an associated *epsilon*, or modifiability parameter. Generally, *epsilon[i][j]* (to unit *i* from unit *j*) is equal to either *lrate* or 0.0, according to whether *weight[i][j]* is modifiable, as specified in the *.net* file. When *lrate* is changed, all the nonzero *epsilon*s are set to *lrate*. However, the user may then independently adjust *epsilon* on any particular weight to any value desired. The user must specify both a receiver and a sender to indicate which *epsilon* to examine or change.

env/ ipattern

Input patterns are specified as a sequence of floating-point numbers, as in earlier programs. The entries "+", ".", and "–" are allowed shorthand for +1.0, 0.0, and −1.0, respectively. Legal values are between 0 and 1, inclusive. If an element of an input pattern has a negative value, the activation of the corresponding input unit is set to the activation of the unit whose index is the absolute value of the negative input pattern element. If the parameter *mu* is nonzero, the activation value of the input unit is incremented by *mu* times its previous value. For example, if element 3 of a particular input pattern is −12, the activation of input unit 3 is set equal to the activation calculated for input unit 12 in processing the previous pattern, plus *mu* times the previous activation of input unit 3.

env/ tpattern

Target patterns are specified as a sequence of floating-point numbers as in **pa**. The entries "+", "–", and "." are interpreted as in the *ipattern*. Legal target values are between 0 and 1, inclusive. If an element of a target pattern has a negative value, then the program acts as though no target was specified for the corresponding output unit; the error for that unit is therefore set to 0.0.

mode/ cascade

By default, the forward pass of processing occurs in a single sweep in **bp**. If this mode variable is set to 1, however, net inputs accumulate gradually over several processing cycles. In this mode the *ncycles* variable determines the number of processing cycles run per pattern tested, and the *cycle* command, following the *test* command, allows the user to continue cycling.

mode/ follow

When this mode switch is set to 1, the **bp** program computes the *gcor* measure, which is the correlation of the gradient in weight space between successive calls to *change_weights*.

mode/ lgrain

Refers to the grain of learning or weight adjustment. By default in the program it is set to *pattern*, which means that weights are adjusted after processing each pattern pair. The *lgrain* variable may also be set to *epoch*, in which case weight changes are accumulated over all patterns presented within an epoch, and then the weights are actually changed only at the end of the epoch.

param/ crate

The cascade rate parameter. Default value is 0.05. Determines the rate of build-up of the net input to each unit on each processing cycle.

param/ lrate

The learning rate parameter. Its default value is 0.05. When *lrate* is reset, all *epsilon*s and *bepsilon*s that are nonzero are reset to the new value of *lrate*.

param/ momentum

The value of the momentum parameter (called *alpha* in *PDP:8*); it has a default value of 0.9. Note that when *lgrain* is set to *pattern*, momentum is building up over patterns; when *lgrain* is *epoch*, momentum is building up over epochs.

param/ mu

This parameter applies to the extension of **bp** to sequential networks. It specifies the extent to which the previous activation of a unit is averaged together with input it receives from other units.

param/ tmax

The target activation actually used when the value given in the target pattern is 1. Defaults to 1.0. The target activation used when the target string contains a 0 is $(1-tmax)$, which is 0.0 when *tmax* is 1.0.

param/ wrange

Range of variability for random weights. If constrained to be positive, the random weights range from 0 to $+wrange$. If constrained to be negative, they range from $-wrange$ to 0. If unconstrained, they range from $-wrange/2$ to $+wrange/2$.

state/ activation

Vector of activation values for units. The ith element corresponds to the activation of the ith unit, as computed during the most recent processing cycle.

state/ bed

Vector of bias error derivative terms.

state/ dbias

Vector of delta bias terms, comparable to delta weights, see below.

state/ delta

Vector of delta terms, or partial derivatives of the error with respect to the net inputs.

state/ dweight

Matrix of delta weights (Δw_{ij} from the equations). This matrix contains the increment last added to the weights, and is used when momentum is nonzero in setting the value of the next increment to add.

state/ error

Vector of error terms, or the partial derivative of the error with respect to a change in the activation of each unit.

state/ gcor

Measures the correlation of the direction of the gradient in weight-space between successive calls to *change_weights*. This is computed only if *follow* mode is on.

state/ netinput

Net input vector.

state/ target

Vector of targets. Note that the ith element corresponds to target value for ith output unit.

state/ wed

Matrix of weight error derivative terms.

OVERVIEW OF EXERCISES

We present three exercises using the basic back propagation procedure. The first one takes you through the XOR problem and is intended to allow you to test and consolidate your basic understanding of the back propagation procedure and the gradient descent process it implements. The second exercise suggests minor variations of the basic back propagation procedure, such as whether weights are changed pattern by pattern or epoch by epoch, and also proposes various parameters that may be explored. The third exercise suggests other possible problems that you might want to explore. Several further exercises are described later in the chapter, in the section on extensions of the back propagation procedure. In Appendix E we provide answers for the questions in Ex. 1.

Ex. 5.1. The XOR Problem

The XOR problem is described at length in *PDP:8*. Here we will be considering one of the two network architectures considered there. This architecture is shown in Figure 9. In this network configuration there are two input units, one for each "bit" in the input pattern; two hidden units; and one output unit. The input units project to the hidden units, and the

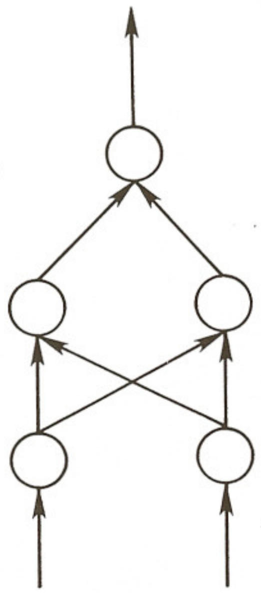


FIGURE 9. Architecture of the XOR network used in Exs. 5.1 and 5.2. (From *PDP:8*, p. 332.)

hidden units project to the output unit; there are no direct connections from the input units to the output units.

All of the relevant files for doing this exercise are contained in the *bp* directory; they are called *xor.tem, xor.str, xor.net*, *xor.pat*, and *xor.wts*. These contain the template, start-up, network, and pattern specifications needed for running the XOR problem.

Once you have your own directory set up with copies of the relevant files, you can start the program. To do so, type

bp xor.tem xor.str

If you wish, you may use a different template file, called *xor2.tem*, instead of *xor.tem*. We describe the case where *xor.tem* is used because the screen layout used in this file is easier to modify for other problems. The layout used in *xor2.tem* should be self-explanatory.

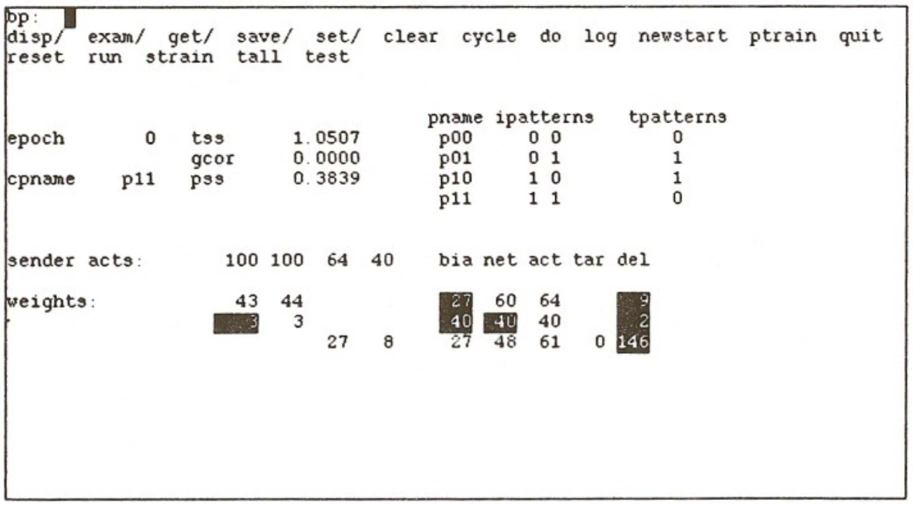
The *xor.str* file is as follows:

```
get network xor.net
get patterns xor.pat
set nepochs 30
set ecrit 0.04
set dlevel 3
set slevel 1
set lflag 1
set mode lgrain epoch
set mode follow 1
set param lrate 0.5
get weights xor.wts
tall
```

This file instructs the program to set up the network as specified in the *xor.net* file and to read the patterns as specified in the *xor.pat* file; it also initializes various variables. Then it reads in an initial set of weights to use for this exercise. Finally, *tall* is called, so that the program processes each pattern. The display that finally appears shows the state of the network at the end of this initial test of all of the patterns. It is shown in Figure 10.

In this figure, below the prompt and the top-level menu and to the left, the current epoch number and the total sum of squares (*tss*) resulting from testing all four patterns are displayed. Also displayed is the *gcor* measure, which is 0.0 at this point since no weight error derivatives have been computed. The next line contains the current pattern number and the sum of squares associated with this pattern. To the right of these entries is the set of input and target patterns for XOR. Below all these entries is a horizontal vector indicating the activations of all of the "sender units." These are units that send their activations forward to other units in the network. The first two are the two input units, and the next two are the two hidden units.

Below this row vector of sender activations is the matrix of weights. The weight in a particular column and row represents the strength of the connection from a particular sender unit indexed by the column to the particular receiver indexed by the row. Note that only the weights that actually



```
bp:
disp/  exam/  get/  save/  set/  clear  cycle  do  log  newstart  ptrain  quit
reset  run  strain  tall  test


                                        pname ipatterns   tpatterns
epoch       0    tss       1.0507        p00     0 0          0
                 gcor      0.0000        p01     0 1          1
cpname    p11    pss       0.3839        p10     1 0          1
                                         p11     1 1          0


sender acts:        100 100   64   40     bia net act tar del

weights:             43  44                27  60  64        9
                      3   3                40  40  40        2
                              27    8      27  48  61    0 146
```

FIGURE 10. The display produced by **bp**, initialized for XOR.

exist in the network are displayed—these are the weights from the two input units to the two hidden units (these are the four numbers below the activations of the two input units) and the weights from the two hidden units to the single output unit (these are the numbers below the activations of the two sending units).

To the right of the weights is a column vector indicating the values of the bias terms for the *receiver* units—that is, all the units that receive input from other units. In this case, the receivers are the two hidden units and the output unit.

To the right of the biases is a column for the net input to each receiving unit. There is also a column for the activations of each of these receiver units. (Note that the hidden units' activations appear twice, once in the row of senders and once in this column of receivers.) The next column contains the target vector, which in this case has only one element since there is only one output unit. Finally, the last column contains the delta values for the hidden and output units.

Note that all activation, weight, and bias values are given in hundredths, so that, for example, 43 means 0.43 and reverse-video 3 means −0.03. For deltas, values are given in *thousandths* so that reverse-video 9 means −0.009.

The display shows what happened when the last pattern pair in the file *xor.pat* was processed. This pattern pair consists of the input pattern (1 1) and the target pattern (0). This input pattern was clamped on the two input units. This is why they both have activation values of 1.0, shown as 100 in the first two entries of the sender activation vector. With these activations of the input units, coupled with the weights from these units to the hidden units, and with the values of the bias terms, the net inputs to the hidden units were set to 0.60 and −0.40, as indicated in the *net* column. Plugging these values into the logistic function, the activation values of 0.64 and 0.40 were obtained for these units. These values are recorded both in the sender activation vector and in the receiver activation vector (labeled *act*, next to the net input vector). Given these activations for the hidden units, coupled with the weights from the hidden units to the output unit and the bias on the output unit, the net input to the output unit is 0.48, as indicated at the bottom of the *net* column. This leads to an activation of 0.61, as shown in the last entry of the *act* column. Since the target is 0.0, as indicated in the target column, the *error*, or (*target* − *activation*) is −0.61; this error, times the derivative of the activation function (that is, *activation* (1 − *activation*)) results in a delta value of −0.146, as indicated in the last entry of the final column. The delta values of the hidden units are determined by back propagating this delta term to the hidden units, as specified by the *compute_error* subroutine.

Q.5.1.1. Show the calculations of the values of *delta* for each of the two hidden units, using the activations and weights as given in this initial screen display. Explain why these values are so small.

At this point, you will notice that the total sum of squares before any learning has occurred is 1.0507. Run another *tall* to understand more about what is happening.

Q.5.1.2. Report the output the network produces for each input pattern and explain why the values are all so similar, referring to the strengths of the weights, the logistic function, and the effects of passing activation forward through the hidden units before it reaches the output units.

Now you are ready to begin learning. Use the *strain* command. This will run 30 epochs of training because the *nepochs* variable is set to 30. If you set *single* to 1, you can follow the *tss* and *gcor* measures as they change from epoch to epoch. You may find in the course of running this exercise that you need to go back and start again. To do this, you should use the *reset* command, followed by

get weights xor.wts

The *xor.wts* file contains the initial weights used for this exercise. This method of reinitializing guarantees that all users will get the same starting weights, independent of possible differences in random number generators from system to system.

Q.5.1.3. The total sum of squares is smaller at the end of 30 epochs, but is only a little smaller. Describe what has happened to the weights and biases and the resulting effects on the activation of the output units. Note the small sizes of the deltas for the hidden units and explain. Do you expect learning to proceed quickly or slowly from this point? Why?

Run another 90 epochs of training (for a total of 120) and see if your predictions are confirmed. As you go along, keep a record of the *tss* at each 30-epoch milestone. (The initial value is given in Figure 10, in case you did not record this previously.) You might find it interesting to observe the results of processing each pattern rather than just the last pattern in the four-pattern set. To do this, you can set the *stepsize* variable to *pattern* rather than the default *epoch*.

At the end of another 60 epochs (total: 180), some of the weights in the network have begun to build up. At this point, one of the hidden units is providing a fairly sensitive index of the number of input units that are on. The other is very unresponsive.

Q.5.1.4. Explain why the more responsive hidden unit will continue to change its incoming weights more rapidly than the other unit over the next few epochs.

Run another 30 epochs. At this point, after a total of 210 epochs, one of the hidden units is now acting rather like an OR unit: its output is about the same for all input patterns in which one or more input units is on.

Q.5.1.5. Explain this OR unit in terms of its incoming weights and bias term. What is the other unit doing at this point?

Now run another 30 epochs. During these epochs, you will see that the second hidden unit becomes more differentiated in its response.

Q.5.1.6. Describe what the second hidden unit is doing at this point, and explain why it is leading the network to activate the output unit most strongly when only one of the two input units is on.

Run another 30 epochs. Here you will see the *tss* drop very quickly.

Q.5.1.7. Explain the rapid drop in the *tss*, referring to the forces operating on the second hidden unit and the change in its behavior. Note that the size of the *delta* for this hidden unit at the end of 270 epochs is about as large in absolute magnitude as the size of the *delta* for the output unit. Explain.

Run the *strain* command one more time. Before the end of the 30 epochs, the value of *tss* drops below *ecrit*, and so *strain* returns. The XOR problem is solved at this point.

Q.5.1.8. Summarize the course of learning, and compare the final state of the weights with their initial state. Can you give an approximate intuitive account of what has happened? What suggestions might you make for improving performance based on this analysis?

Ex. 5.2. Variations With XOR

There are several further studies one can do with XOR. You can study the effects of varying:

- One of the parameters of the model (*lrate, wrange, momentum*).
- The learning grain (*lgrain* [*epoch* vs. *pattern*]).
- The training regime (permuted vs. sequential test; this makes a difference only when *lgrain* is equal to *pattern*).

- The starting configuration (affected by *wrange*, but also by the particular random values of the weights).
- The number of hidden units.
- Whether particular weights are constrained to be positive, negative, or linked to each other.

The possibilities are almost endless, and all of them have effects. For this exercise, pick one of these possible dimensions of variation, and run at least three more runs, comparing the results to those you obtained in Ex. 5.1.

Q.5.2.1. Describe what you have chosen to vary, how you chose to vary it, and what results you obtained in terms of the rate of learning, the evolution of the weights, and the eventual solution achieved.

Where relevant, it is useful to try several different values along the dimension you have chosen to vary, performing several runs with different initial starting weights in each instance and using the *newstart* command to get a new value for the random seed before each run. You might want to try the same starting configuration with different values on the dimension you are varying. This can be done using *reset* instead of *newstart*. For example, suppose you are examining the effects of varying *lrate*. You can issue the *newstart* command to get a random weight set to use with one value of *lrate*, then use *reset* to try the same starting configuration again, after using the *set/ param/ lrate* command to adjust the value of *lrate*.

If you want to constrain weights to be positive, negative, or linked, you need only change the *.net* file a little, replacing *r*'s with *p*'s or *n*'s for weights that you want to force to be positive or negative. It is best to make a new *.net* file, with a new name, and a new *.str* file that includes the *get/ network* command to read in this new *.net* file.

If you choose to vary the number of hidden units, you will have to modify the *.net* and *.tem* files. Here we give very briefly a checklist of what needs to be done to accomplish this. For the network file, you must increase *nunits* in the definitions. You must also alter the network part of the file and the biases part of the file to specify the values of the connections and bias terms involving the added units. Since *xor.net* uses the % convention, you will have to understand how it works, as described in Appendix C. You have to change the %-lines to reflect the new hidden units.

For the template file, you have to make more space in the layout for the additional rows and columns. The layout gives four character positions for each column in the weight matrix, so if you add three new hidden units, you have to add spaces to move everything that is to the right of the weight matrix further to the right by 12 columns. Also, for each added hidden unit, you have to add an additional line in the layout. These extra lines in the template file go just above the last line that has $'s on it. Both these things are done by adding blank lines and spaces to the layout. You must also alter the template specifications themselves, specifying the correct starting element and number of elements for the vector displays and specifying the correct starting row, number of rows, starting column, and number of columns for the weights.

Ex. 5.3. Other Problems for Back Propagation

Construct a different problem to study, either choosing from those discussed in *PDP:8* or choosing a problem of your own. Set up the appropriate network, template, pattern, and start-up files, and experiment with using back propagation to learn how to solve your problem.

Q.5.3.1. Describe the problem you have chosen, and why you find it interesting. Explain the network architecture that you have selected for the problem and the set of training patterns that you have used. Describe the results of your learning experiments. Evaluate the back propagation method for learning and explain your feelings as to its adequacy, drawing on the results you have obtained in this experiment and any other observations you have made from the readings or from this exercise.

Hints. Try not to be too ambitious. You will learn a lot even if you limit yourself to a network with 10 units. For an easy way out, you could choose to study the 4-2-4 encoder problem described in *PDP:8*. The files *424.tem, 424.str, 424.net,* and *424.pat* are already set up for this problem.

EXTENSIONS OF BACK PROPAGATION

Up to this point we have discussed the use of back propagation in one-pass, feedforward networks only. Here we discuss three extensions. The first maintains the strictly feedforward character of the networks we have been considering but provides for gradual build-up of activation. The second simulates recurrent activation in a limited way by using linked connections. The third introduces real recurrence of activation by allowing the units in the network to send their outputs back to specified input units. We will briefly consider each of these extensions and offer an exercise to go with each.

CASCADED FEEDFORWARD NETWORKS

One of the precursors of our work on PDP models was a model called the *cascade* model (McClelland, 1979). This model was a purely linear, feedforward, multilayer network. Units at each level took on activations based on inputs from the preceding level according to the following equation:

$$a_{ir}(t) = k_r \sum_j w_{ij} a_{js}(t) + (1 - k_r) a_{ir}(t-1) \tag{2}$$

Here r and s index some receiving level and the level sending to it, i and j index units within levels, and k_r is a rate constant governing the rate at which the activations of units at level r reach the value that the summed input is driving them toward.

Such a system has interesting temporal properties and provides a useful framework for accounting for many aspects of reaction time data (see McClelland, 1979, for details), but its computational capabilities are seriously limited. In particular, with linear units, a multilayer system can always be replaced by a single layer, and as we have noted repeatedly, there are limits on the computations that can be performed in a single layer. To overcome these limitations, multiple layers of units, with some form of nonlinear activation function, are required.

The idea we describe in this section preserves the desirable computational characteristics of the nonlinear networks we have been considering in this chapter but combines with these the gradual build-up of activation characteristic of the cascade model. To achieve this, we introduce one change into the cascade equation: Instead of directly setting the activation of units on the basis of this equation, we use it to determine the net input; the activation is then calculated from the net input using the logistic function. Thus the equation for the net input to a unit becomes

$$net_{ir}(t) = k_r \sum_j w_{ij} a_{js}(t) + (1 - k_r) net_{ir}(t-1), \tag{3}$$

and the equation for its activation is

$$a_{ir}(t) = \frac{1}{1 + e^{-net_{ir}(t)}}. \tag{4}$$

In our implementation of this scheme, there is a single rate parameter called *crate* (for cascade rate) that is used for all units rather than a separate rate parameter for each level.

One very nice feature of this scheme is that if an input pattern comes on at time $t = 0$ and stays on, the *asymptotic* activation each unit reaches is identical to the activation that it reaches in a single step in the standard, one-pass feedforward computation used up to now in back propagation.

Thus, we can view the one-pass feedforward computation as one that computes the asymptotic activations that would result from a process that may in real systems be gradual and continuous. We can then train the network to achieve a particular asymptotic activation value for each of several patterns and then observe its dynamical properties.

Before we actually turn to an exercise in which we do just what we have described, we mention one more characteristic of these cascaded feedforward networks: Their dynamical properties depend on the initial state. Here we assume that the initial state is the pattern of activation that results from processing an input pattern consisting of all-zero activations for all of the input units. For this to work, the network must in general be trained to produce some appropriate output state for this initial input state. Here we simply assume that the desired initial output state is also all-zero activations on all of the output units.

Ex. 5.4. A Cascaded XOR Network

After training the XOR network of Ex. 5.1 in the standard way, turn on *cascade* mode and, using the *test* command, examine the time course of activation of the hidden and output units for the patterns in the XOR pattern set.

To run this exercise, run the program with the *cas.tem* and *cas.str* files. These differ only slightly from the standard XOR files. The *.tem* file displays the cycle number (which is 0 until *cascade* mode is turned on). The *.str* file sets *nepochs* to 300 and sets *ncycles* to 100. After start-up, issue the *strain* command, which will cause the network to learn the solution to the XOR problem from the first exercise, finishing at epoch 289, where the *tss* falls below 0.04. Then enter *set/ mode/ cascade 1* to turn on *cascade* mode. In this mode, the activations of the units are initialized to the asymptotic values they would have for the input (0 0), which, conveniently, was in the training set with target output (0). Now use the *test* command to test each of the three nonzero patterns (patterns 1, 2, and 3). The *test* command will automatically set *stepsize* to *cycle* and turn on *single* mode, so you will be able to study the time course of activation step by step. To obtain a simple graph of the results after the run is over, you can use the *log* command to open a log file before you begin testing. Then you can use the *plot* program described in Appendix D. Once a log file has been opened with the *log* command, the *.str* and *.tem* files set up the *dlevels* of the templates and the global *slevel* of the program so that the pattern name, the cycle number, and the activations of the hidden and output units are logged on each cycle.

Q.5.4.1. Explain why the output unit is initially more strongly activated by the (1 1) input pattern than by either the (1 0) or the (0 1) patterns, and explain why the activation eventually "turns around" for the (1 1) pattern, following an overall U-shaped activation curve.

RECURRENT NETWORKS

In *PDP:8*, it is noted that it is possible to emulate a recurrent network in which units feed activation to themselves and to units that project to them by essentially duplicating each unit. Here we consider a simple three-unit example of such a network.

Conceptually, the network we will consider consists of three units, with each unit having a modifiable bias term and a modifiable connection to itself and to each other unit. Actually, though, the network consists of three layers of three units each. Each unit in the first layer projects to each unit in the second layer, and each unit in the second layer projects to each unit in the third layer. Units in the second and third layer also have modifiable bias terms. Connections from the first layer to the second layer are linked to the corresponding connections from the second to the third layer. Similarly, the bias terms of the units in the second layer are linked to the corresponding bias terms in the third layer. These links force corresponding weights and biases to be equal, so there is really only one set of weights and one set of biases, as there would be in our conceptual network. This three-layer network allows us to supply an initial input pattern to the input units and to simulate two time steps of the activation process that would occur in our conceptual three-unit network.

Ex. 5.5. The Shift Register

In this exercise we will train the three-unit recurrent network to be a shift register. The task is to learn to take an arbitrary pattern of 1s and 0s on the three input units and shift that pattern two bits to the right. This is done by training the network with input-output pairs, each consisting of one of the eight 3-bit binary input vectors paired with the same vector shifted two bits right. Actually, it is appropriate to think of the units as forming a ring so that when a bit is shifted from the right-most position it appears in the left-most position.

The exercise is carried out by calling the **bp** program with the files *rec.tem* and *rec.str*. The display shows the activation of each of the three copies of the three units, along with standard information such as the *tss*, and so on. The eight different binary patterns of three bits have been read in as input patterns, along with the same patterns shifted two bits to the right as the corresponding targets. If you run a *tall* at this point, you will see that the output of the third stage of processing is close to 0.5 for all eight of the different input patterns.

The weights and bias terms have been set up as already described, and the bias terms are additionally constrained to be negative. The other parameters were chosen arbitrarily. The problem is an easy one to learn, as you can prove to yourself by running the *strain* command. The network will generally find a solution in about 40 to 200 epochs. Once the solution has been found, you can run a *tall* and see how the network has solved the problem. If you wish to see the weights, biases, and target patterns in addition to the activation vectors, you can set *dlevel* to 3. In this state, the display will show the weight matrix twice, once with the initial set of activations feeding into it from above with the intermediate set to the left and once with the second set feeding into it with the third set to its left.

Q.5.5.1. Does the network always find the same solution to the problem? Describe all the different solutions you get after repeated runs, using *newstart* to reinitialize the network between runs. What happens if you try removing the constraints that force the biases to be negative and link the weights together? Do these changes increase the number of different solutions?

Hints. To remove the negative constraints, you simply have to set up a new *.net* file, deleting the word *negative* wherever it appears in the *rec.net* file. To remove the links, you can replace the letters *a* to *i* in the *network* specification and *j* to *l* in the *biases* specification with *r*.

SEQUENTIAL NETWORKS

A somewhat different type of recurrent network—here called a sequential network—has been proposed by Jordan (1986). He devised this type of network originally for generating a sequence of outputs, such as phonemes, as would occur, for example, in saying a word.

In Jordan's formulation, the network consists of a set of input units called *plan* units and another set of input units called *current-state* units. Both sets feed into a set of hidden units, which in turn feed into a set of output, or *next-state*, units. This type of network is illustrated in Figure 11. At the beginning of a sequence, a plan pattern is input to the plan units, and the current-state units are set to 0. Feedforward processing occurs, as in standard back propagation, producing the first output pattern (e.g., the first phoneme of the word specified in the plan). This output pattern is then copied back to the current-state units for the next feedforward sweep. Actually, the activation of each current-state unit at the beginning of the next processing sweep is set equal to the activation of the corresponding

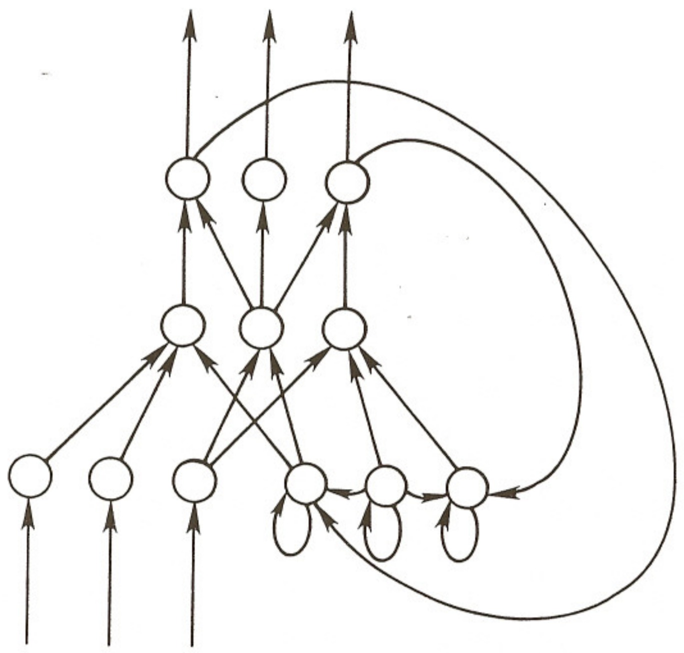


FIGURE 11. Basic structure of Jordan's sequential networks. (From "Attractor Dynamics and Parallelism in a Connectionist Sequential Machine" by M. I. Jordan, 1986, *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*. Hillsdale, NJ: Lawrence Erlbaum Associates. Copyright 1986 by M. I. Jordan. Reprinted by permission.)

output unit plus μ (*mu*) times the unit's previous activation. This gives the current-state units a way of capturing the prior history of activation in the network.

We have provided a simple facility for implementing Jordan's sequential nets as well as many more complex variants. This facility employs negative integers in input patterns to indicate that the unit's activation should be derived from activations produced on the previous cycle. The particular negative integer indicates which unit's activation should be used in setting the activation of the input unit. Thus, if element i of a particular input pattern is equal to $-j$, the activation of unit i will be set to $a_j + \mu a_i$, where a_i and a_j are taken to be the activations of units i and j from the previous cycle. The parameter *mu* can be set using the *set/ param/ mu* command (its value should be between 0 and 1).

To implement Jordan's sequential networks using this scheme, we first set up a network, say, with four plan units, four current-state units, four hidden units, and four next-state units. Then we set up each sequence we want to train the network to learn as a set of pattern pairs. In each pattern pair belonging to the same sequence, the plan field stays the same. The target patterns are the sequence of desired outputs. For the first pattern in the sequence, the input values for the current state units are 0. For subsequent patterns in the sequence, the input values for the current state units are set to $-n$, where n is the index of the corresponding output unit. Thus, the patterns to specify that the network should interpret the plan pattern (1 0 1 0) as an instruction to turn on first the first output unit, then the second, then the third, and then the fourth would be as follows:

```
Plan             Current State           Target
1 0 1 0           0   0   0   0          1 0 0 0
1 0 1 0         -12 -13 -14 -15          0 1 0 0
1 0 1 0         -12 -13 -14 -15          0 0 1 0
1 0 1 0         -12 -13 -14 -15          0 0 0 1
```

(In the pattern file each pattern would also have a name at the beginning of the line.) Note that the assumptions made by Jordan can be seen as a special case of a very general class of sequential models made possible by the facility to set the activation of input units based on prior activations. Inputs can be set to have activations based on the activations of hidden units as well as output units, and external input can be intermixed at will with feedback. It is even possible to set up any desired combination of previous inputs by hard-wiring hidden units to receive inputs from particular sets of input units.[6]

Ex. 5.6. Plan-Dependent Sequence Generation

To illustrate the use of sequential networks, we provide a simple exercise in which a back propagation network is trained to turn on the output units in sequence, either left to right if the pattern over the plan units is (1 0 1 0) or right to left if the pattern is (0 1 0 1). As in the example just described, the network has four plan units, four current-state units, four hidden units, and four output, or next-state, units. The network is trained with the four patterns from each of the two desired sequences, for a total of eight training patterns.

To run this exercise, start up the **bp** program with the files *seq.tem* and *seq.str*. The activations of the plan units and current-state units will be shown in a vertical column on the left of the screen, the activations of the hidden units to the right of these, and the activations of the output units to the right of these. Before training, run *tall* to see that the output is weak and random, then run *strain* until the network solves the problem (it

[6] The only restriction on the use of the previous-state facility is that an input unit cannot receive its input from the previous activation of a lower-numbered unit. In practice this should not be much of a restriction since lower-numbered units would always be other input units.

generally takes several hundred epochs). Once the problem is solved (*tss* less that 0.1), do a *tall* again to verify that the network does as instructed.

Q.5.6.1. See if your can figure out some of the reasons why this task is harder than the XOR task. If you want to explore this kind of network further, you might want to study the effects of various parameters on learning time. For example, you might want to experiment with different values of *mu*, *lrate*, and *momentum*.

Hints. In thinking about the first part of the question, you might note what happens to the patterns of activation on the output units, and therefore the current-state units, during the early parts of training. In your explorations for the second part of the question, the most interesting effects occur with variations in *mu*. See if you can discover and understand these effects.