Learning Rules for Multilayer Feedforward Neural Networks

This chapter extends the gradient-descent-based delta rule of Chapter 2 to multilayer feedforward neural networks. The resulting learning rule is commonly known as *error backpropagation* (or *backprop),* and it is one of the most frequently used learning rules in many applications of artificial neural networks.

The backprop learning rule is central to much current work on learning in artificial neural networks. In fact, the development of backprop is one of the main reasons for the renewed interest in artificial neural networks. Backprop provides a computationally efficient method for changing the weights in a feedforward network, with differentiable activation function units, to learn a training set of input-output examples. Backprop-trained multilayer neural nets have been applied successfully to solve some difficult and diverse problems, such as pattern classification, function approximation, nonlinear system modeling, time-series prediction, and image compression and reconstruction. For these reasons, most of this chapter is devoted to the study of backprop, its variations, and its extensions.

Backpropagation is a gradient-descent search algorithm that may suffer from slow convergence to local minima. In this chapter, several methods for improving back-prop's convergence speed and avoidance of local minima are presented. Whenever possible, theoretical justification is given for these methods. A version of backprop based on an enhanced criterion function with global search capability is described which, when properly tuned, allows for relatively fast convergence to good solutions.

Consider the two-layer feedforward architecture shown in Figure 0-1. This network receives a set of scalar signals $\{x_0, x_1, ..., x_n\}$ where x_0 is a bias signal equal to 1. This set of signals constitutes an input vector $\mathbf{x} \in R^{n+1}$. The layer receiving the input signal is called the *hidden layer*. Figure 0-1 shows a hidden layer having *J* units. The output of the hidden layer is a (*J* + l)-dimensional real-valued vector $\mathbf{z} = [z_0, z_1, ..., z_J]^{\mathrm{T}}$. Again, $z_0 = 1$ represents a bias input and can be thought of as being generated by a "dummy" unit (with index zero) whose output z_0 is clamped at 1. The vector **z** supplies the input for the *output layer* of *L* units. The output layer generates an *L*-dimensional vector

y in response to the input **x** which, when the network is fully trained, should be identical (or very close) to a "desired" output vector **d** associated with **x**.

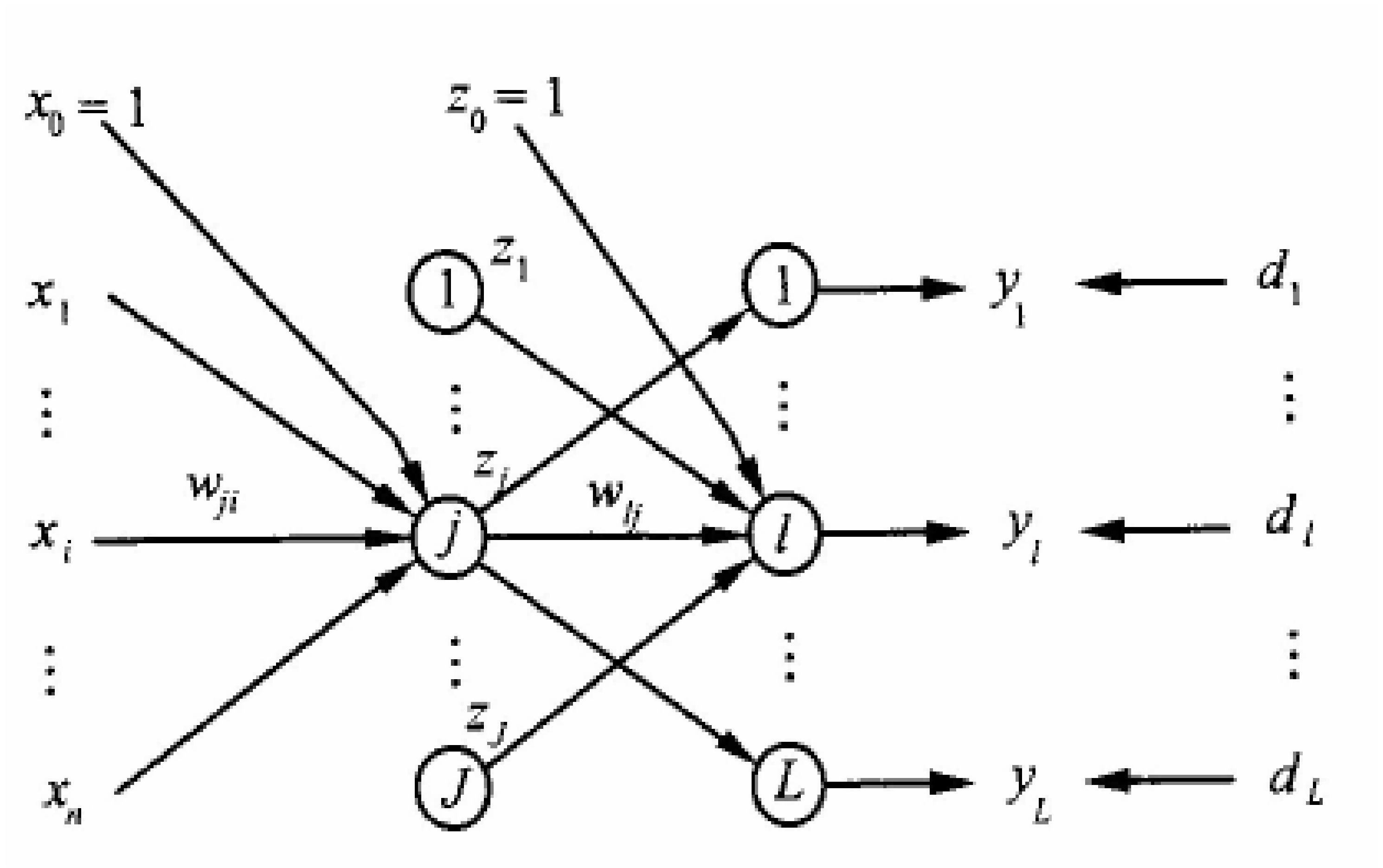

Figure 0-1 A two-layer fully interconnected feedforward neural network architecture. For clarity, only selected connections are drawn.

The activation function f_h of the hidden units is assumed to be a differentiate nonlinear function [typically, f_h is the logistic function defined by $f_h(net) = 1/\left(1 + e^{-\lambda\, net}\right)$, or hyperbolic tangent function $f_h(net) = \tanh(\beta\, net)$, with values for λ and β close to unity]. Each unit of the output layer is assumed to have the same activation function, denoted f_0 the functional form of f_0 is determined by the desired output signal/pattern representation or the type of application. For example, if the desired output is real valued (as in some function approximation applications), then a linear activation $f_h(net) = \lambda\, net$ may be used. On the other hand, if the network implements a pattern classifier with binary outputs, then a saturating nonlinearity similar to f_h may be used for f_0. In this case, the components of the desired output vector **d** must be chosen within the range of f_0. It is important to note that if f_h is linear, then one can always collapse the net in Figure 0-1 to a single-layer net and thus lose the universal approximation/mapping capabilities discussed in Chapter 4. Finally, we denote by w_{ji}, the weight of the *j*th hidden unit associated with the input signal x_i. Similarly, w_{lj} is the weight of the *l*th output unit associated with the hidden signal z_j.

Next, consider a set of m input/output pairs $\left\{\mathbf{x}^k, \mathbf{d}^k\right\}$, where $\mathbf{d}^k$ is an L-dimensional vector representing the desired network output upon presentation of $\mathbf{x}^k$. The objective here is to adaptively adjust the $J(n + 1) + L(J + 1)$ weights of this network such that the underlying function/mapping represented by the training set is approximated or learned. Since the learning here is supervised (i.e., target outputs are available), an error function may be defined to measure the degree of approximation for any given setting of the network's weights. A commonly used error function is the SSE measure, but this is by no means the only possibility, and later in this chapter, several other error functions will be discussed. Once a suitable error function is formulated, learning can be viewed (as was done in Chapters 2) as an optimization process. That is, the error function serves as a criterion function, and the learning algorithm seeks to minimize the criterion function over the space of possible weight settings. For instance, if a differentiable criterion function is used, gradient descent on such a function will naturally lead to a learning rule.

$$E(\mathbf{w}) = \frac{1}{2}\sum_{l=1}^{L}\left(d_l - y_l\right)^2 \qquad (3.1)$$

Here, **w** represents the set of all weights in the network. Note that Equation (3.1) is the "instantaneous" SSE criterion of Equation (2.32) generalized for a multiple-output network.

3.1 Error Backpropagation Learning Rule

Since the targets for the output units are explicitly specified, one can use the delta rule directly, derived in Section 2.3 for updating the w_{lj} weights. That is,

$$\Delta w_{lj} = w_{lj}^{\text{new}} - w_{lj}^{c} = -\rho_0 \frac{\partial E}{\partial w_{lj}} = \rho_o \left(d_l - y_l \right) f_o'\left(net_l \right) z_j \qquad (3.2)$$

with $l = 1, 2, \ldots, L$ and $j = 0, 1, \ldots, J$. Here $net_l = \sum_{j=0}^{J} w_{lj} z_j$ is the weighted sum for the *l*th output unit, f_o' is the derivative of f_o, with respect to *net*, and w_{lj}^{new} and w_{lj}^{c} represent the updated (new) and current weight values, respectively. The z_j values are computed by propagating the input vector **x** through the hidden layer according to

$$z_j = f_h\left(\sum_{i=0}^{n} w_{ji} x_i\right) = f_h\left(net_j\right) \qquad j = 1, 2, \ldots, J \qquad (3.3)$$

The learning rule for the hidden-layer weights w_{ji} is not as obvious as that for the output layer because we do not have available a set of target values (desired outputs) for hidden units. However, one may derive a learning rule for hidden units by attempting to minimize the output-layer error. This amounts to propagating the output errors $\left(d_l - y_l\right)$ back through the output layer toward the hidden units in an attempt to estimate "dynamic" targets for these units. Such a learning rule is termed *error backpropagation* or the *backprop learning rule* and may be viewed as an extension of the delta rule [Equation (3.2)] used for updating the output layer. To complete the derivation of backprop for the hidden-layer weights, and similar to the preceding derivation for the output-layer weights, gradient descent is performed on the criterion function in Equation (3.1), but this time, the gradient is calculated with respect to the hidden weights:

$$\Delta w_{ji} = -\rho_h \frac{\partial E}{\partial w_{ji}} \qquad j = 1, 2, \ldots, J; \quad i = 0, 1, 2, \ldots, n \qquad (3.4)$$

where the partial derivative is to be evaluated at the current weight values. Using the chain rule for differentiation, one may express the partial derivative in Equation (3.4) as

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial z_j} \frac{\partial z_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}} \qquad (3.5)$$

with

$$\frac{\partial net_j}{\partial w_{ji}} = x_i \qquad (3.6)$$

$$\frac{\partial z_j}{\partial net_j} = f_h'\left(net_j\right) \qquad (3.7)$$

and

$$\frac{\partial E}{\partial z_j} = \frac{\partial}{\partial z_j}\left\{\frac{1}{2}\sum_{l=1}^{L}\left[d_l - f_o\left(net_l\right)\right]^2\right\}$$

$$= -\sum_{l=1}^{L}\left[d_l - f_o\left(net_l\right)\right]\frac{\partial f_o\left(net_l\right)}{\partial z_j} \tag{3.8}$$

$$= -\sum_{l=1}^{L}\left(d_l - y_l\right) f_o'\left(net_l\right) w_{lj}$$

Now, upon substituting Equations (3.6) through (3.8) into Equation (3.5) and using Equation (3.4), the desired learning rule is obtained:

$$\Delta w_{ji} = \rho_h \left[\sum_{l=1}^{L}\left(d_l - y_l\right) f_o'\left(net_l\right) w_{lj}\right] f_h'\left(net_j\right) x_i \tag{3.9}$$

By comparing Equation (3.9) with Equation (3.2), one can immediately define an "estimated target" d_j for the *j*th hidden unit implicitly in terms of the backpropagated error signal as follows:

$$d_j - z_j \triangleq \sum_{l=1}^{L}\left(d_l - y_l\right) f_o'\left(net_l\right) w_{lj} \tag{3.10}$$

It is usually possible to express the derivatives of the activation functions in Equations (3.2) and (3.9) in terms of the activations themselves. For example, for the logistic activation function,

$$f'(net) = \lambda f(net)\left[1 - f(net)\right] \tag{3.11}$$

and for the hyperbolic tangent function,

$$f'(net) = \beta\left[1 - f^2(net)\right] \tag{3.12}$$

These learning equations may also be extended to feedforward nets with more than one hidden layer and/or nets with connections that jump over one or more layers. The complete procedure for updating the weights in a feedforward neural net utilizing these rules is summarized below for the two-layer architecture of Figure 0-1. This learning procedure will be referred to as *incremental backprop* or just *backprop.*

1. Initialize all weights and refer to them as "current" weights w_{lj}^{c} and w_{ji}^{c} (see Section 3.3.1 for details).

2. Set the learning rates ρ_o and ρ_h to small positive values (refer to Section 3.3.2 for additional details).

3. Select an input pattern $\mathbf{x}^k$ from the training set (preferably at random) and propagate it through the network, thus generating hidden- and output-unit activities based on the current weight settings.

4. Use the desired target $\mathbf{d}^k$ associated with $\mathbf{x}^k$, and employ Equation (3.2) to compute the output layer weight changes Δw_{lj}.

5. Employ Equation (3.9) to compute the hidden-layer weight changes Δw_{ji}. Normally, the current weights are used in these computations. In general, enhanced error correction may be achieved if one employs the updated output-layer weights $w_{lj}^{\text{new}} = w_{lj}^{c} + \Delta w_{lj}$. However, this comes at the added cost of recomputing y_l and $f_o'(net_l)$.

6. Update all weights according to $w_{lj}^{\text{new}} = w_{lj}^{c} + \Delta w_{lj}$ and $w_{ji}^{\text{new}} = w_{ji}^{c} + \Delta w_{ji}$ for the output and hidden layers, respectively.

7. Test for convergence. This is done by checking some preselected function of the output errors[1] to see if its magnitude is below some preset threshold. If convergence is met, stop; otherwise, set $w_{ji}^{c} = w_{ji}^{\text{new}}$ and $w_{lj}^{c} = w_{lj}^{\text{new}}$, and go to step 3. It should be noted that backprop may fail to find a solution that passes the convergence test. In this case, one may try to reinitialize the search process, tune the learning parameters, and/or use more hidden units.

This procedure is based on *incremental learning*, which means that the weights are updated after every presentation of an input pattern. Another alternative is to employ *batch learning*, where weight updating is performed only after all patterns (assuming a finite training set) have been presented. Batch learning is formally stated by summing the right-hand sides of Equations (3.2) and (3.9) over all patterns $\mathbf{x}^k$ This amounts to gradient descent on the criterion function

[1] A convenient selection is the root-mean-square (RMS) error given by $\sqrt{2E/(mL)}$ with E as in Equation (3.13). An alternative, and more sensible stopping test may be formulated by using cross-validation; see Section 3.3.6 for details.

$$E(\mathbf{w}) = \frac{1}{2}\sum_{k=1}^{m}\sum_{l=1}^{L}(d_l - y_l)^2 \qquad (3.13)$$

Even though batch updating moves the search point **w** in the direction of the true gradient at each update step, the "approximate" incremental updating is more desirable for two reasons: (1) it requires less storage, and (2) it makes the search path in the weight space stochastic (here, at each time step, the input vector **x** is drawn at random), which allows for a wider exploration of the search space and, potentially, leads to better-quality solutions. When backprop converges, it converges to a local minimum of the criterion function. This fact is true of any gradient-descent-based learning rule when the surface being searched is nonconvex; i.e., it admits local minima. Using stochastic approximation theory, Finnoff (1994) showed that for "very small" learning rates (approaching zero), incremental backprop approaches batch backprop and produces essentially the same results. However, for small constant learning rates there is a nonnegligible stochastic element in the training process that gives incremental backprop a quasi-annealing character in which the cumulative gradient is continuously perturbed, allowing the search to escape local minima with small and shallow basins of attraction. Thus solutions generated by incremental backprop are often practical ones. The local minima problem can be eased further by heuristically adding random noise to the weights (von Lehman et al., 1988) or by adding noise to the input patterns (Sietsma and Dow, 1988). In both cases, some noise-reduction schedule should be employed to dynamically reduce the added noise level toward zero as learning progresses.

Example 3.1 Consider the two-class problem shown in Figure 0-2. The points inside the shaded region belong to class **B**, and all other points are in class **A**. A three-layer feedforward neural network with backprop training is employed that is supposed to learn to distinguish between these two classes. The network consists of an eight-unit first hidden layer, followed by a second hidden layer with four units, followed by a one-unit output layer. Such a network is said to have an 8-4-1 architecture. All units employ a hyperbolic tangent activation function. The output unit should encode the class of each input vector, a positive output indicates class **B** and a negative output indicates class **A**. Incremental backprop was used with learning rates set to 0.1. The training set consists of 500 randomly chosen points, 250 from region **A** and another 250 from region **B**. In this training set, points representing class **B** and class **A** were assigned desired output (target) values of +1 and –1, respectively[2]. Training was performed for several hundred cycles over the training set.

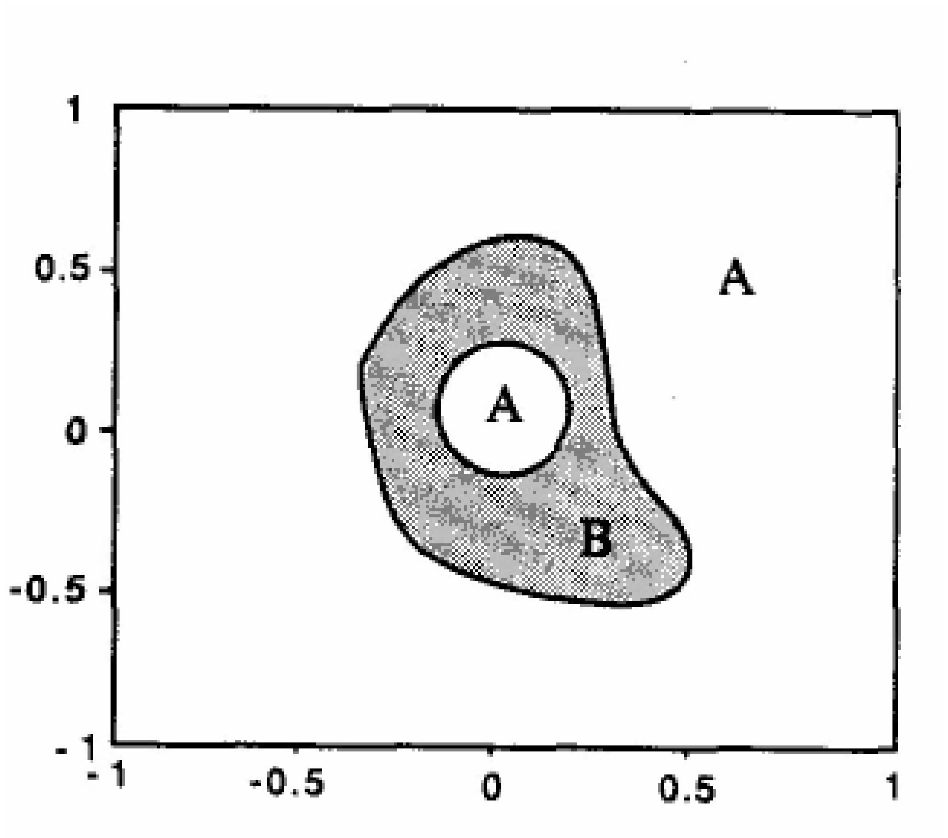

Figure 0-2 Decision regions for the pattern-classification problem in Example 3.1

[2] In fact, the actual targets used were offset by a small positive constant ε (say, $\varepsilon = 0.1$) away from the limiting values of the activation function. This resulted in replacing the +1 and -1 targets by $1-\varepsilon$ and $-1+\varepsilon$, respectively. Otherwise, backprop tends to drive the weights of the network to infinity and thereby slow the learning process.

Figure 0-3 shows geometric plots of all unit responses upon testing the network with a new set of 1000 uniformly (randomly) generated points inside the $[-1,+1]^2$ region. In generating each plot, a black dot was placed at the exact coordinates of the test point (input) in the input space if and only if the corresponding unit response was positive. The boundaries between the dotted and the white regions in the plots represent approximate decision boundaries learned by the various units in the network. Figure 0-3*a-h* represents the decision boundaries learned by the eight units in the first hidden layer. Figure 0-3*i-l* shows the decision boundaries learned by the four units of the second hidden layer. Figure 0-3*m* shows the decision boundary realized by the output unit. Note the linear nature of the separating surface realized by the first-hidden-layer units, from which complex nonlinear separating surfaces are realized by the second-hidden-layer units and ultimately by the output-layer unit. This example also illustrates how a single-hidden-layer feedforward net (counting only the first two layers) is capable of realizing convex, concave, as well as disjoint decision regions, as can be seen from Figure 0-3*i-l*. Here, we neglect the output unit and view the remaining net as one with an 8-4 architecture.

The present problem can also be solved with smaller networks (fewer numbers of hidden units or even a network with a single hidden layer). However, the training of such smaller networks with backprop may become more difficult. A smaller network with a 5-3-1 architecture utilizing a variant backprop learning procedure has a comparable separating surface to the one in Figure 0-3*m*.

Huang and Lippmann (1988) employed Monte Carlo simulations to investigate the capabilities of backprop in learning complex decision regions (see Figure 4.5). They reported no significant performance difference between two- and three-layer feedforward nets when forming complex decision regions using backprop. They also demonstrated that backprop's convergence time is excessive for complex decision regions and that the performance of such trained classifiers is similar to that obtained with the *k*-nearest neighbor classifier (Duda and Hart, 1973). Villiers and Barnard (1993) reported similar simulations but on data sets that consisted of a "distribution of distributions" where a typical class is a set of clusters (distributions) in the feature space; each of which can be more or less spread out and which might involve some or all of the dimensions of the feature space; the distribution of distributions thus assigns a probability to each distribution in the data set. It was found for networks of equal complexity (same number of weights) that there is no significant difference between the quality of "best" solutions generated by two- and three-layer backprop-trained feedforward networks; actually, the two-layer nets demonstrated better performance, on average. As for the speed of convergence, three-layer nets converged faster if the number of units in the two hidden layers were roughly equal.

Gradient-descent search may be eliminated altogether in favor of a stochastic global search procedure that guarantees convergence to a global solution with high probability; genetic algorithms and simulated annealing are examples of such procedures. However, the assured (in probability) optimality of these global search procedures comes at the expense of slow convergence. Next, a deterministic search procedure termed *global descent* is presented that helps backprop reach globally optimal solutions.

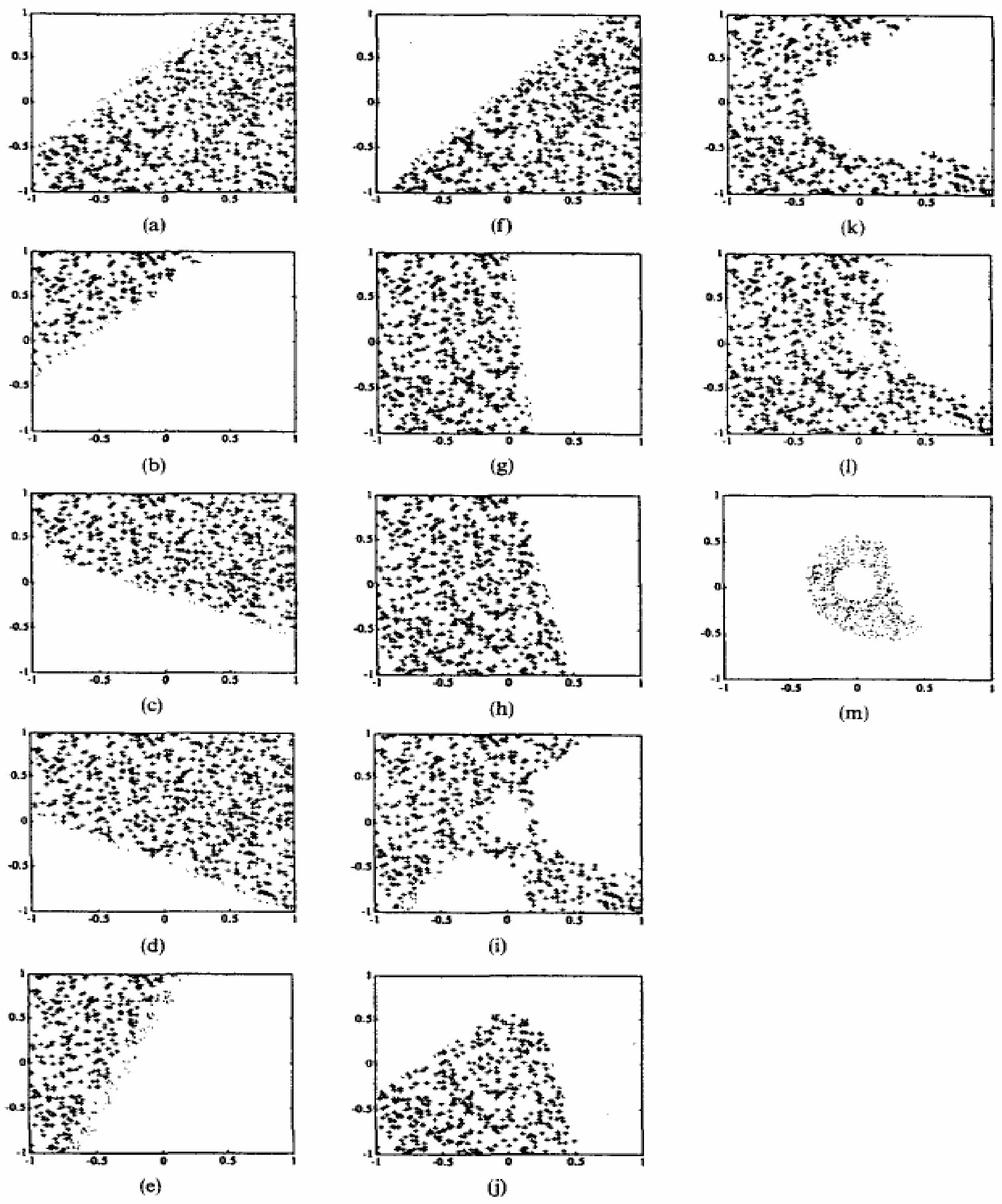

Figure 0-3 Separating surfaces generated by the various units in the 8-4-1 network of Example 3.1. *(a-h)* Separating surfaces realized by the units in the first hidden layer; (*i*-*l*) separating surface realized by the units in the second hidden layer, (*m*) separating surface realized by the output unit.