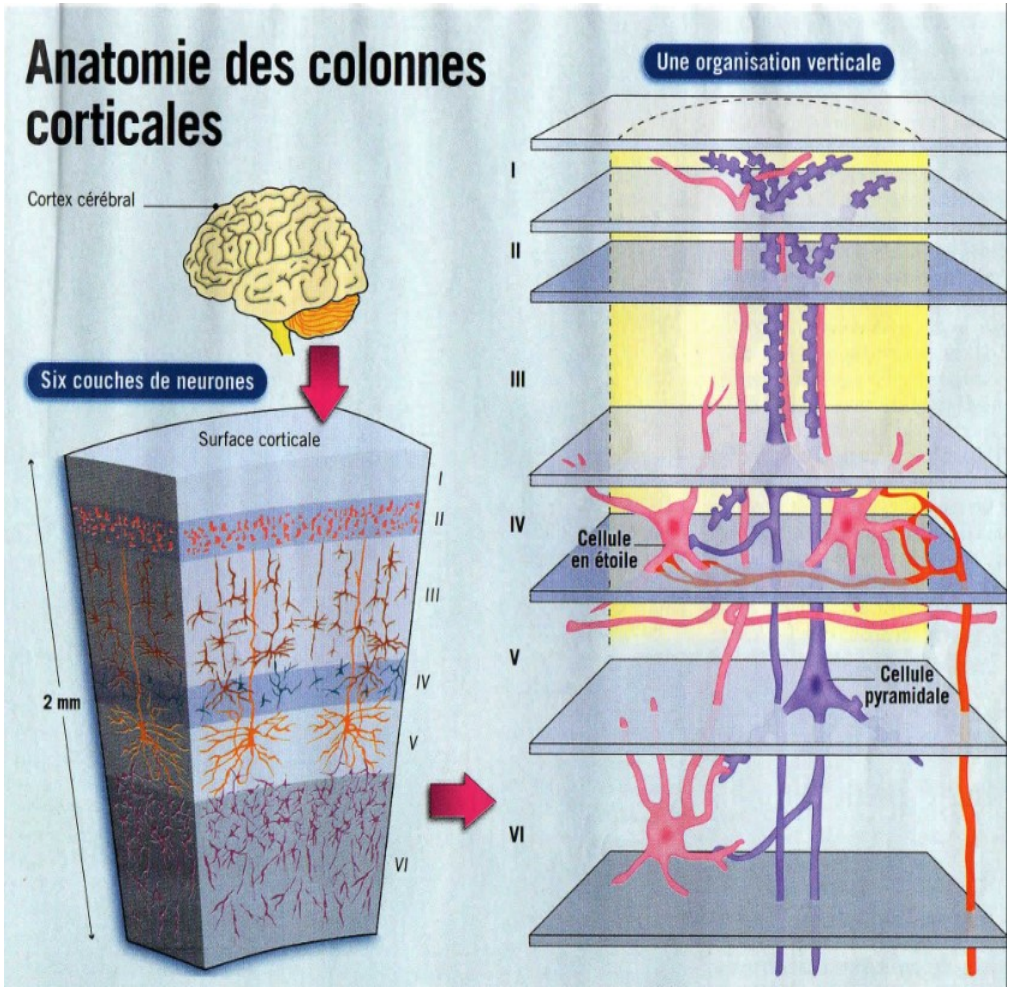
Overview of Unit 4

- **Multilayer Network**
- **Error Backpropagation Learning Rule**
- **Computation Capabilities of MLP Networks**

Multilayer Networks



The cerebral cortex (or neocortex) is the pleated layer about 2 mm thick covering the cerebral hemispheres and providing superior cognitive functions. To humans (and mice), the cortex is composed of six distinct layers of neurons (cutting diagram left). These horizontal layers are distinguished by the type of neurons present (pyramidal, granular interneurons, etc.), but also by the type of connections they establish with neighboring or more distant ones. Functional studies have shown that these neurons are also organized vertically (right diagram) into functional units called "cortical columns". Each column or cylinder cortex contains about 10,000 neurons that receive and process a certain type of sensory stimuli (such as the orientation of an object or the frequency of a sound). The human brain would have a million of these cortical columns connected to each other not only in the neighborhood but also at a great distance between brain areas or the other hemisphere. This organization has been demonstrated in columns for the primary visual and auditory cortex then generalized to the entire neocortex.

Neuron simulation integrated into the cortex which electrical activity can be visualized through different colors.

Consider the two-layer feedforward architecture shown in Figure Error! No text of specified style in document.**-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 Error! No text of specified style in document.**-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**.

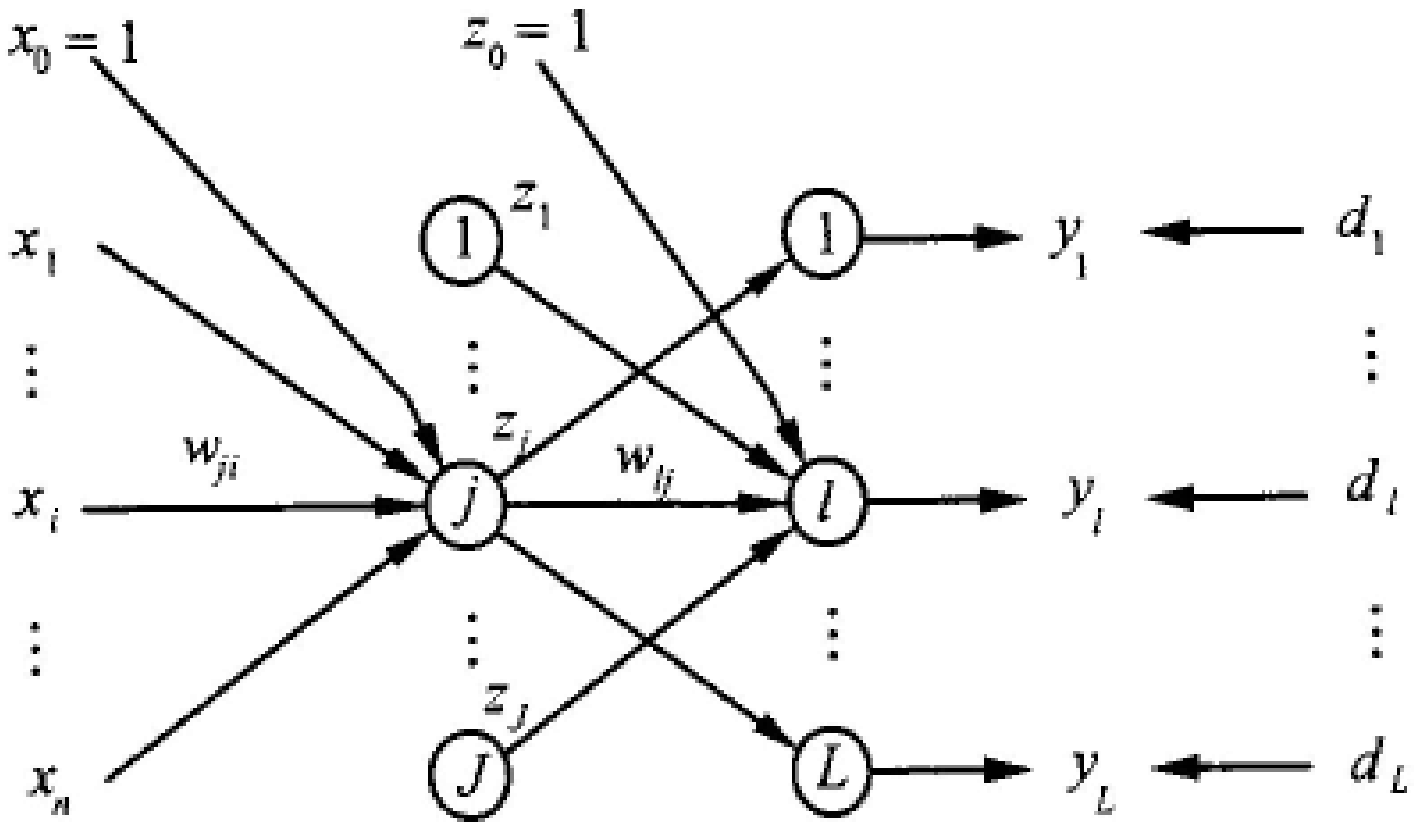


Figure Error! No text of specified style in document.**-1** A two-layer fully interconnected feedforward neural network architecture. For clarity, only selected connections are drawn.

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.

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 $\{\mathbf{x}^k, \mathbf{d}^k\}$, 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 several other error functions can be envisaged.

Once a suitable error function is formulated, learning can be viewed (as was done in Unit 3) as an optimization process. That is, the error function serves as a loss / 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 supervised learning rule for adjusting the weights w_{ji} and w_{lj} such that the following error function is minimized (in a local sense) over the given training set.

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

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

Error Backpropagation Learning Rule

Since the targets for the output units are explicitly specified, one can use the delta rule directly, derived in Unit 2 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 $l = 1,2,...,L$ and $j = 0,1,...,J$

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

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

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:

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

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) and obtain, finally, the desired learning rule

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

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

$$d_j - z_j \square \sum_{l=1}^{L} \ldots y_l) f_o'(net_l) w_{lj} \tag{3.10}$$

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 Error! No text of specified style in document.**-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.

2. Set the learning rates ρ_o and ρ_h to small positive values.

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.

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.

[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.

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

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

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 for any gradient-descent-based learning rule when the surface being searched is nonconvex; i.e., it admits local minima.

Example 3.1 Consider the two-class problem shown in Figure Error! No text of specified style in document.**-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.

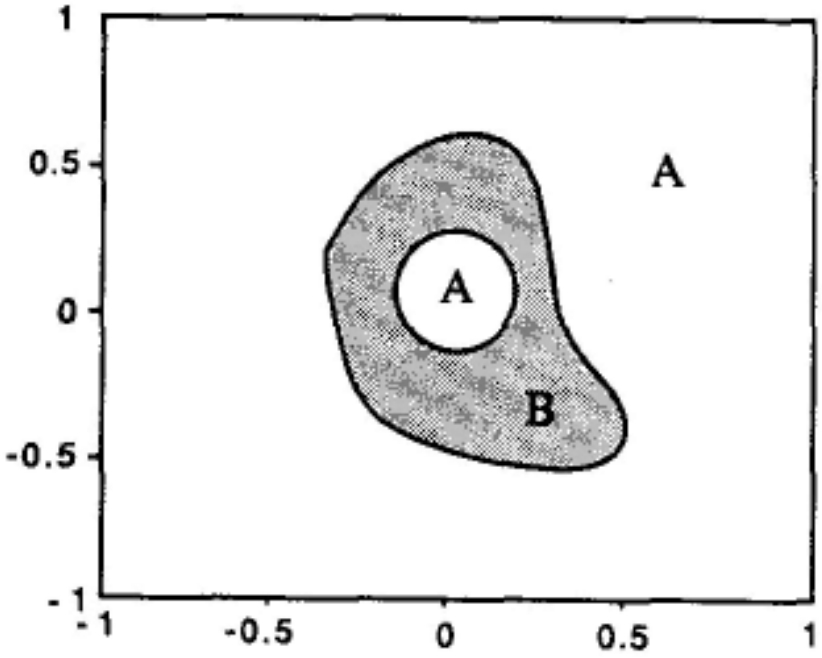


Figure Error! No text of specified style in document.**-2** Decision regions for the pattern-classification problem in Example 3.1

[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 Error! No text of specified style in document.-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 Error! No text of specified style in document.-3*a-h* represents the decision boundaries learned by the eight units in the first hidden layer.

Figure Error! No text of specified style in document.-3*i-l* shows the decision boundaries learned by the four units of the second hidden layer.

Figure Error! No text of specified style in document.-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 Error! No text of specified style in document.-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 Error! No text of specified style in document.-3*m*.

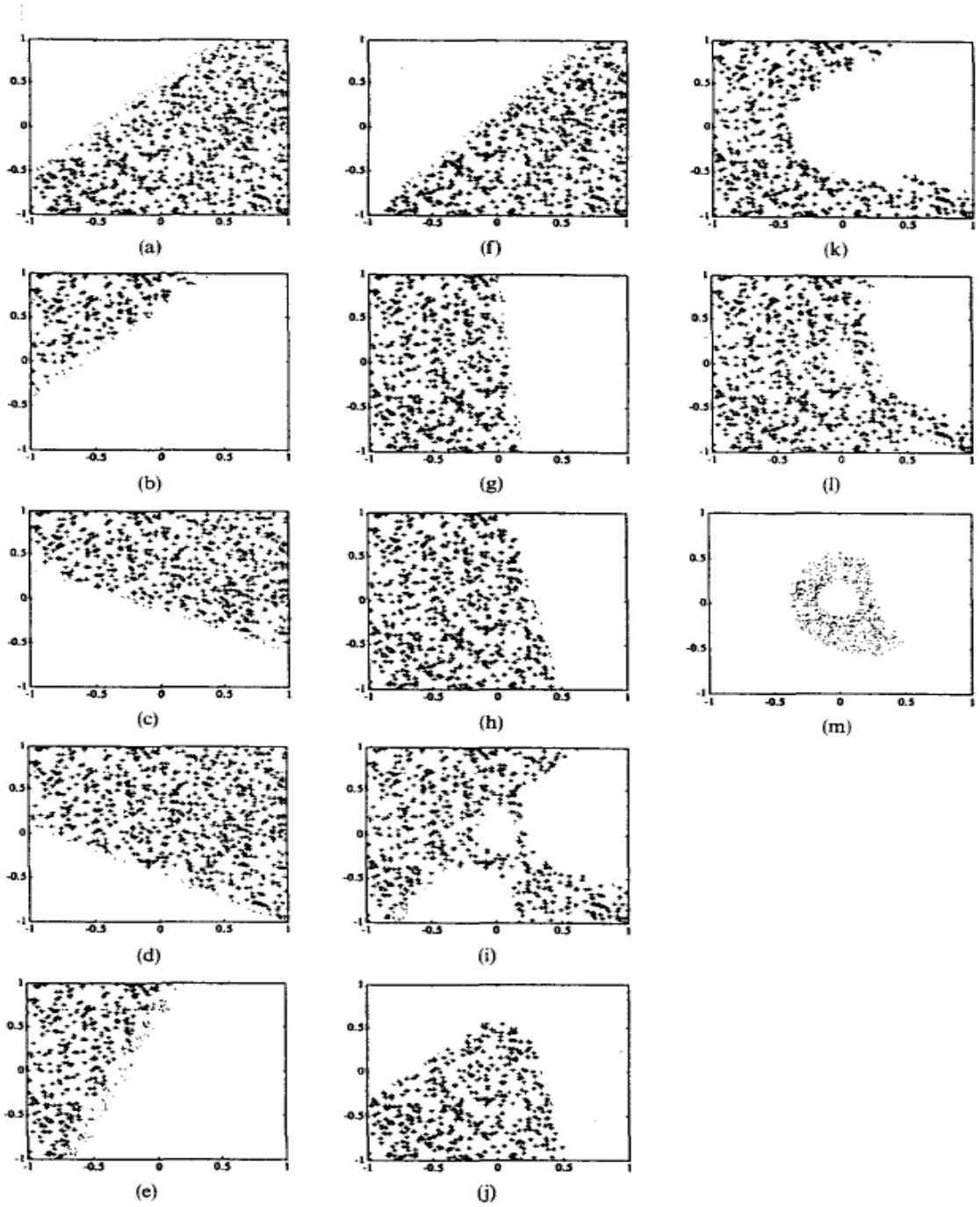
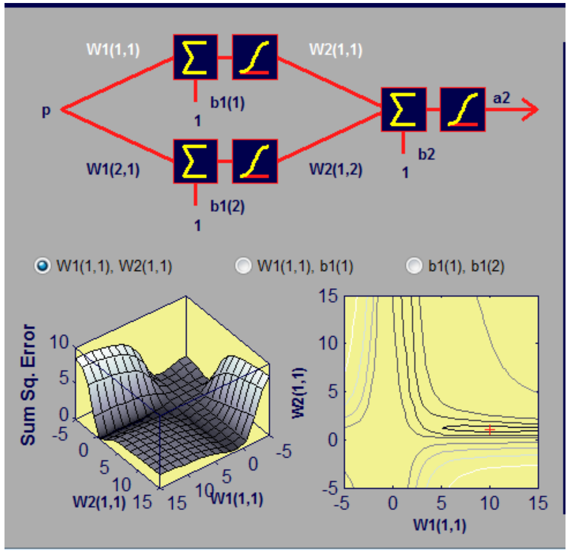
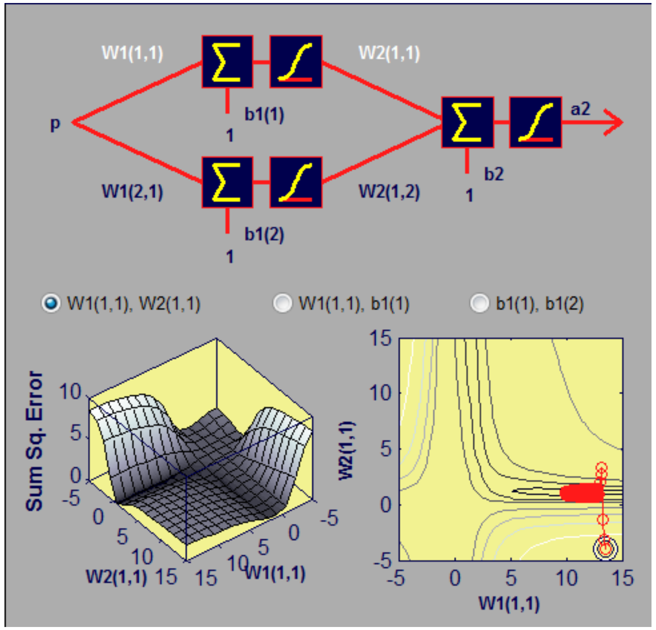
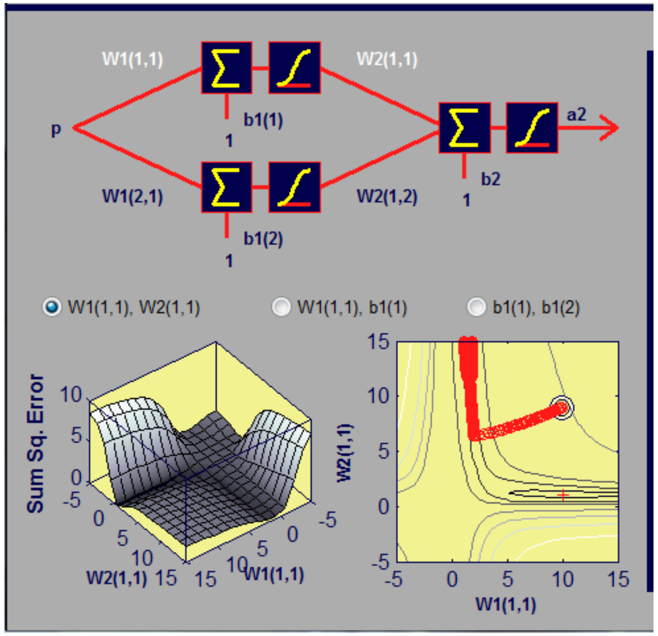


Figure Error! No text of specified style in document.**-3** Separating surfaces generated by the various units in the 8-4-1 network of Example 3.1. *(a-h)* Separating surfaces realized by the units in the first hidden layer; (*i-l*) separating surface realized by the units in the second hidden layer, (*m*) separating surface realized by the output unit.

D


W1(1,1)
W2(1,1)
p
b1(1)
a2
W1(2,1)
W2(1,2)
b2
b1(2)
W1(1,1), W2(1,1)
W1(1,1), b1(1)
b1(1), b1(2)
Sum Sq. Error
W2(1,1)
W1(1,1)


W1(1,1)
W2(1,1)
p
b1(1)
a2
W1(2,1)
W2(1,2)
b2
b1(2)
W1(1,1), W2(1,1)
W1(1,1), b1(1)
b1(1), b1(2)
Sum Sq. Error
W2(1,1)
W1(1,1)


W1(1,1)
W2(1,1)
p
b1(1)
a2
W1(2,1)
W2(1,2)
b2
b1(2)
W1(1,1), W2(1,1)
W1(1,1), b1(1)
b1(1), b1(2)
Sum Sq. Error
W2(1,1)
W1(1,1)

Computational Capabilities of MLP

Since one of our motivations for considering multilayer networks is to avoid the perceptron's limitation to linear decision regions, the first question that arises is

"What kinds of decision regions can a multilayer network represent?"

Perhaps surprisingly, the answer is that

with just three layers and enough units in each layer, a multilayer network can approximate any decision rule. In fact, the third layer (the output layer) consists of just a single output unit, so multiple units are needed only in the first and second layer.

For decision rules that assign to class 1 a possibly infinite number of areas in the plane (or volumes in N-space), this result can be shown relatively simply as follows.

First, we can approximate any decision rule that assigns feature vectors in some convex set to class 1 and all others to class 0. (A set of points is convex if a line between any two points in the set is always entirely within the set.) This is because a convex set can be approximated by a polyhedron obtained by intersecting a number of half-spaces (Figures 10.2 and 10.3).

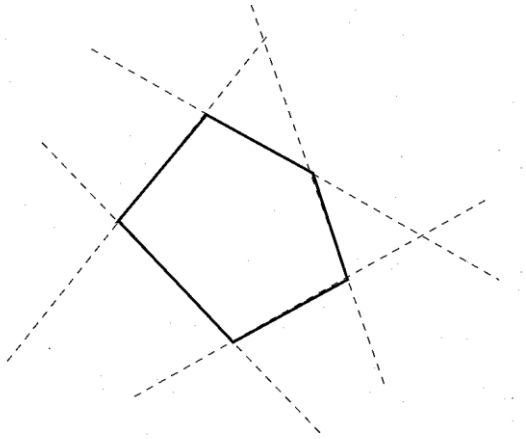


Figure 10.2 Creating a polyhedron by intersecting half-spaces.

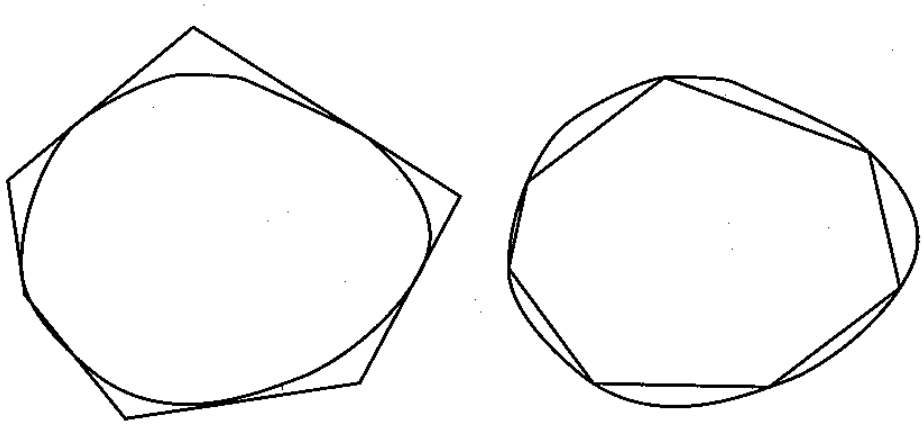


Figure 10.3 Two common ways to approximate a convex set by a polyhedron.

Each half-space has a linear decision boundary that can be implemented by a single perceptron**.** The various half-spaces used to approximate the convex set are implemented by different units all in the first layer.

To get the intersection of the half-planes, we pass the outputs of the perceptrons in the first layer to a single perceptron in the second layer that computes the logical AND of its inputs, so that the output is 1 if and only if every input to the single perceptron in the second layer is 1.

This happens exactly when the feature vector lies inside the polyhedron defined by the half-spaces.

By taking more units in the first layer, we get a polyhedron with more faces, and therefore, a better approximation to the convex set.

To approximate an arbitrary (nonconvex) set, we can first approximate it with a union of convex sets (Figure 10.4).

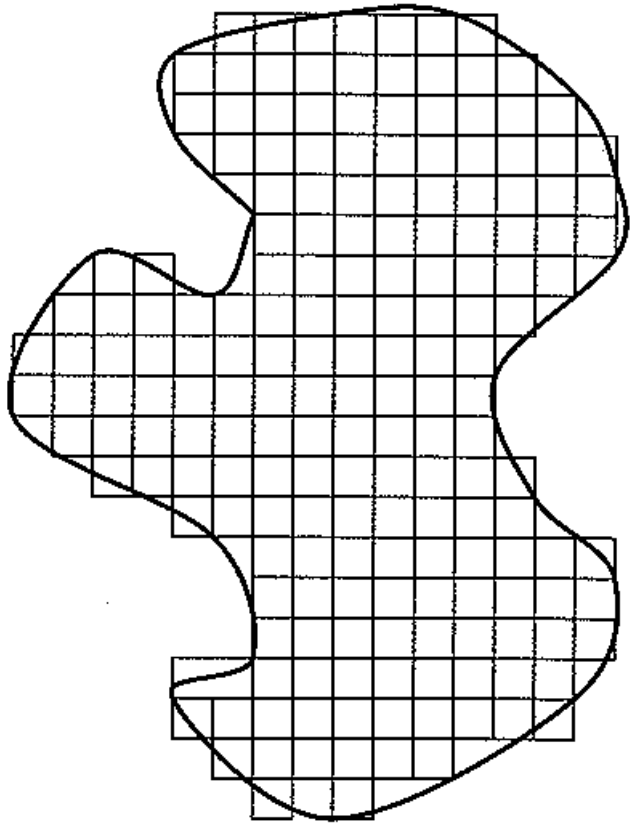


Figure 10.4 Approximating a nonconvex set with hypercubes.

One systematic way to do this is to partition the space regularly along each dimension to form hypercubes. Each of the constituent convex sets can be approximated as described above, so we only need a way to take the union of these sets. To get the union, we can use a single perceptron that computes the logical OR of its inputs.

Finally, the network used to approximate a general set has the following form.

- Each unit in the first layer computes a half-space.
- The outputs of this layer are passed to the second layer, where each unit perform the logical AND of those half-spaces needed to approximate various convex sets.
- The outputs from the second layer are then passed through a final unit that performs an OR operation in order to take the union of those convex sets. (See Figure 10.5)

<u>NOTE</u> The below network can't generalize because it recognize, indeed without error, only the Figure 10.4

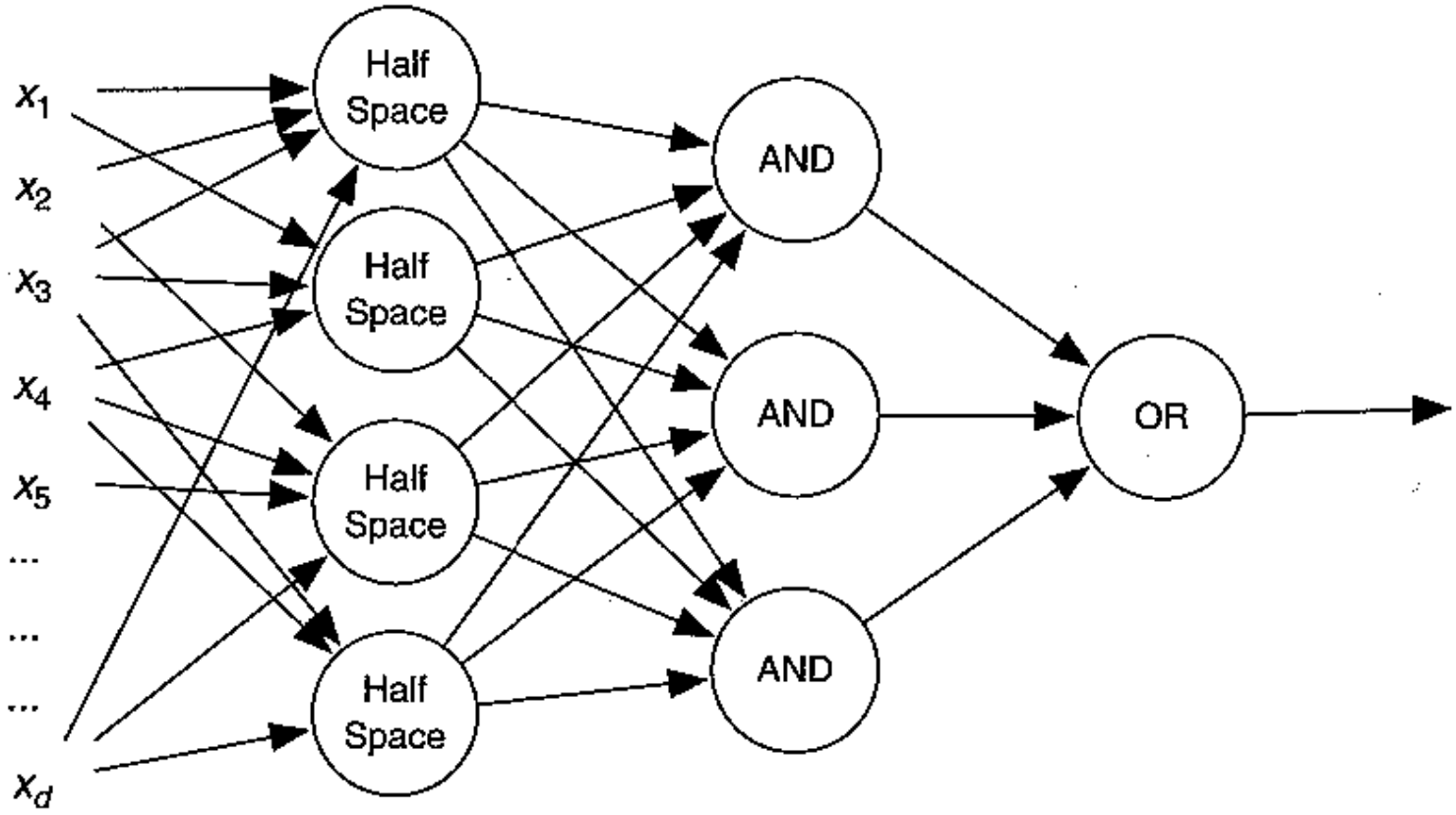


Figure 10.5 Network structure to approximate general sets.