C H A P T E R

6

The Counterpropagation Network

The Counterpropagation network (CPN) is the most recently developed of the models that we have discussed so far in this text. The CPN is not so much a new discovery as it is a novel combination of previously existing network types. Hecht-Nielsen synthesized the architecture from a combination of a structure known as a *competitive* network and Grossberg's *outstar* structure [5, 6]. Although the network architecture, as it appears in its originally published form in Figure 6.1, seems rather formidable, we shall see that the operation of the network is quite straightforward.

Given a set of vector pairs, $(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_L, \mathbf{y}_L)$, the CPN can learn to associate an x vector on the input layer with a y vector at the output layer. If the relationship between x and y can be described by a continuous function, Φ, such that $\mathbf{y} = \Phi(\mathbf{x})$, the CPN will learn to approximate this mapping for any value of x in the range specified by the set of training vectors. Furthermore, if the inverse of Φ exists, such that x is a function of y, then the CPN will also learn the inverse mapping, $\mathbf{x} = \Phi^{-1}(\mathbf{y})$.[1] For a great many cases of practical interest, the inverse function does not exist. In these situations, we can simplify the discussion of the CPN by considering only the forward-mapping case, $\mathbf{y} = \Phi(\mathbf{x})$.

In Figure 6.2, we have reorganized the CPN diagram and have restricted our consideration to the forward-mapping case. The network now appears as

[1] We are using the term *function* in its strict mathematical sense. If y is a function of x, then every value of x corresponds to one and only one value of y. Conversely, if x is a function of y, then every value of y corresponds to one and only one value of x. An example of a function whose inverse is not a function is, $y = x^2$, $-\infty < x < \infty$. A somewhat more abstract, but perhaps more interesting, situation is a function that maps images of animals to the name of the animal. For example, "CAT" = Φ("picture of cat"). Each picture represents only one animal, but each animal corresponds to many different pictures.

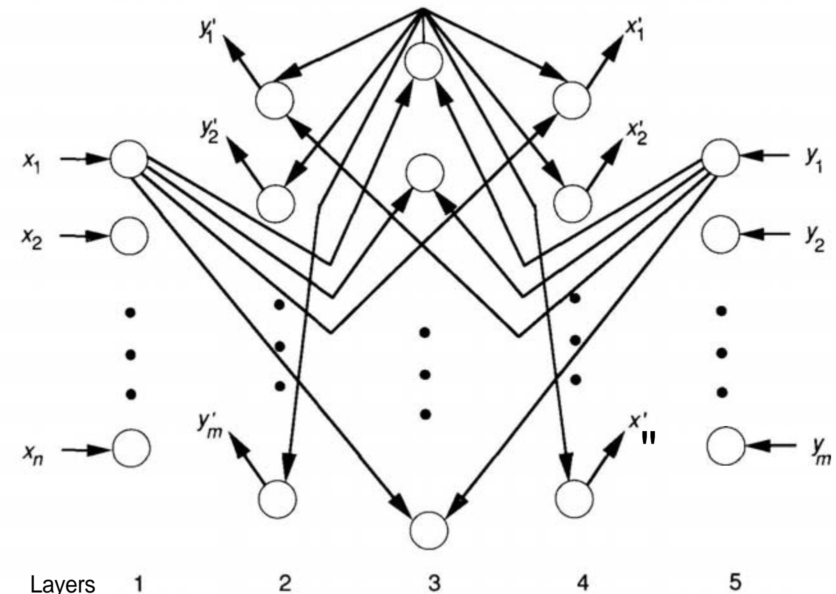


Figure 6.1 This spiderlike diagram of the CPN architecture has five layers: two input layers (1 and 5), one hidden layer (3), and two output layers (2 and 4). The CPN gets its name from the fact that the input vectors on layers 1 and 2 appear to propagate through the network in opposite directions. *Source: Reprinted with permission from Robert Hecht-Nielsen, "Counterpropagation networks." In* Proceedings of the IEEE First International Conference on Neural Networks. *San Diego, CA, June 1987. ©1987 IEEE.*

a three-layer architecture, similar to, but not exactly like, the backpropagation network discussed in Chapter 3. An input vector is applied to the units on layer 1. Each unit on layer 2 calculates its net-input value, and a competition is held to determine which unit has the largest net-input value. That unit is the only unit that sends a value to the units in the output layer. We shall postpone a detailed discussion of the processing until we have examined the various components of the network.

CPNs are interesting for a number of reasons. By combining existing network types into a new architecture, the CPN hints at the possibility of forming other, useful networks from bits and pieces of existing structures. Moreover, instead of employing a single learning algorithm throughout the network, the CPN uses a different learning procedure on each layer. The learning algorithms allow the CPN to train quite rapidly with respect to other networks that we have studied so far. The tradeoff is that the CPN may not always yield sufficient

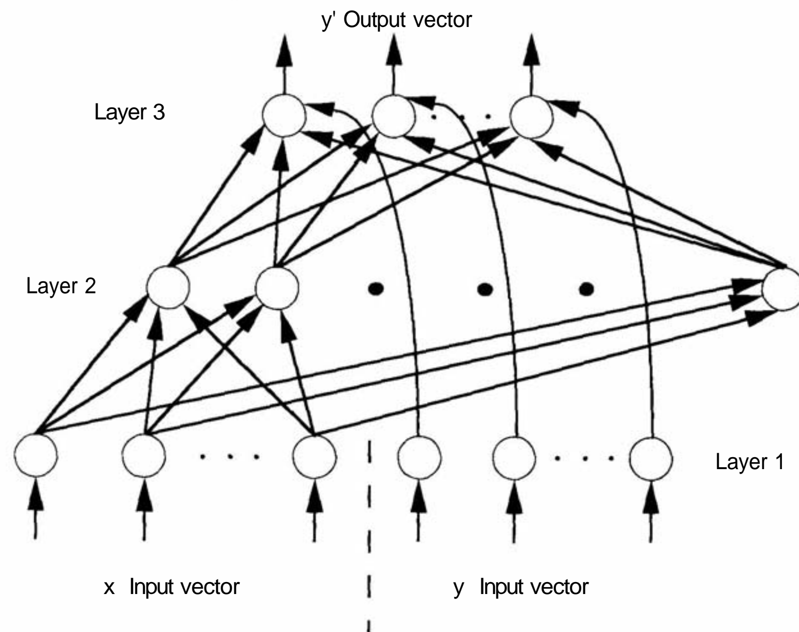


Figure 6.2 The forward-mapping CPN is shown. Vector pairs from the training set are applied to layer 1. After training is complete, applying the vectors (x, 0) to layer 1 will result in an approximation, $\mathbf{y}'$, to the corresponding y vector, at the output layer, layer 3. See Section 6.2 for more details of the training procedure.

accuracy for some applications. Nevertheless, the CPN remains a good choice for some classes of problems, and it provides an excellent vehicle for rapid prototyping of other applications. In the next section, we shall examine the various building blocks from which the CPN is constructed.

6.1 CPN BUILDING BLOCKS

The PEs and network structures that we shall study in this section play an important role in many of the subsequent chapters in this text. For that reason, we present this introductory material in some detail. There are four major components: an input layer that performs some processing on the input data, a processing element called an **instar,** a layer of instars known as a **competitive network,** and a structure known as an **outstar.** In Section 6.2, we shall return to the discussion of the CPN.

6.1.1 The Input Layer

Discussions of neural networks often ignore the input-layer processing elements, or consider them simply as pass-through units, responsible only for distributing input data to other processing elements. Computer simulations of networks usually arrange for all input data to be scaled or normalized to accommodate calculations by the computer's CPU. For example, input-value magnitudes may have to be scaled to prevent overflow error during the sum-of-product calculations that dominate most network simulations. Biological systems do not have the benefits of such preprocessing; they must rely on internal mechanisms to prevent saturation of neural activities by large input signals. In this section, we shall examine a mechanism of interaction among processing elements that overcomes this **noise-saturation dilemma** [2]. Although the mechanism has some neurophysiological plausibility, we shall not examine any of the biological implications of the model.

Examine the layer of processing elements shown in Figure 6.3. There is one input value, I_i, for each of the n units on the layer. The total input pattern intensity is given by $7 = \sum_i I_i$. Corresponding to each I_i, we shall define a quantity

$$\Theta_i = I_i\left(\sum_i I_i\right)^{-1} \tag{6.1}$$

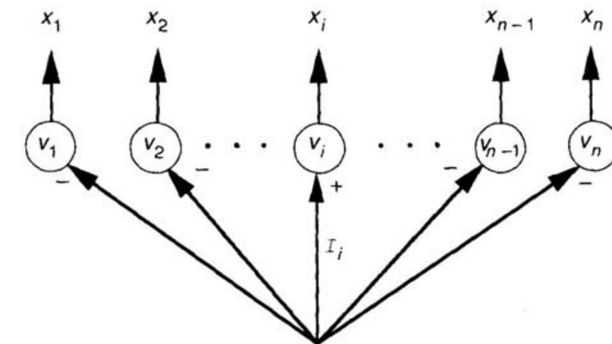


Figure 6.3 This layer of input units has n processing elements, $\{v_1, v_2, \ldots, v_n\}$. Each input value, I_i, is connected with an excitatory connection (arrow with a plus sign) to its corresponding processing element, v_i. Each Ii is connected also to every other processing element, v_k, $k \neq i$, with an inhibitory connection (arrow with a minus sign). This arrangement is known as *on-center, off-surround.* The output of the layer is proportional to the normalized reflectance pattern.

The vector, $(\Theta_1, \Theta_2, \ldots, \Theta_n)^t$, is called a **reflectance pattern.** Notice that this pattern is normalized in the sense that $\sum_i \Theta_i = 1$.

The reflectance pattern is independent of the total intensity of the corresponding input pattern. For example, the reflectance pattern corresponding to the image of a person's face would be independent of whether the person were being viewed in bright sunlight or in shade. We can usually recognize a familiar person in a wide variety of lighting conditions, even if we have not seen her previously in the identical situation. This experience suggests that our memory stores and recalls reflectance patterns.

The outputs of the processing elements in Figure 6.3 are governed by the set of differential equations,

$$\dot{x}_i = -Ax_i + (B - x_i)I_i - x_i \sum_{k \neq i} I_k \tag{6.2}$$

where $0 < x_i(0) < B$, and $A, B > 0$. Each processing element receives a net excitation (on-center) of $(B - x_i)I_i$ from its corresponding input value, I_i. The addition of inhibitory connections (off-surround), $-x_i I_k$, from other units is responsible for preventing the activity of the processing element from rising in proportion to the absolute pattern intensity, I_i.

Once an input pattern is applied, the processing elements quickly reach an equilibrium state ($\dot{x}_i = 0$) with

$$x_i^{eq} = \Theta_i \frac{BI}{A + I} \tag{6.3}$$

where we have used the definition of Θ_i in Eq. (6.1). The output pattern is normalized, since

$$\sum_i x_i = \frac{BI}{A + I} = B\left(\frac{A}{I} + 1\right)^{-1}$$

which is always less than B. Thus, the pattern of activity that develops across the input units is proportional to the reflectance pattern, rather than to the original input pattern.

After the input pattern is removed, the activities of the units do not remain at their equilibrium values, nor do they return immediately to zero. The activity pattern persists for some time while the term $-Ax_i$ reduces the activities gradually back to a value of zero.

An input layer of the type discussed in this section is used for both the x-vector and y-vector portions of the CPN input layer shown in Figure 6.1. When performing a digital simulation of the CPN, we can simplify the program by normalizing the input vectors in software. Whether the input-pattern normalization is accomplished using Eq. (6.2), or by some preprocessing in software, depends on the particular implementation of the network.

Exercise 6.1:

1. Solve Eq. (6.2) to find $x_i(t)$ explicitly, assuming $x_i(0) = 0$ and a constant input pattern, I.
2. Assume that the input pattern is removed at $t = t'$, and find $x_i(t)$ for $t > t'$.
3. Draw the graph of $x_i(t)$ from $t = 0$ to some $t \gg t'$. What determines how quickly $x_i(t)$ (a) reaches its equilibrium value, and (b) decays back to zero after the input pattern is removed?

Exercise 6.2: Investigate the equations

$$x_i = -Ax_i + (B - x_i)I_i$$

as a possible alternative to Eq. (6.2) for the input-layer processing elements. For a constant reflectance pattern, what happens to the activation of each processing element as the total pattern intensity, I, increases?

Exercise 6.3: Consider the equations

$$\pm i = -Ax_i + (B - x_i)I_i - (x_i + C)\sum_{k \neq i} I_k$$

which differ from Eq. (6.2) by an additional inhibition term, $C \sum_{k \neq i} I_k$:

1. Suppose $I_i = 0$, but $\sum_{k \neq i} Ik > 0$. Show that x_i can assume a negative value. Does this result make sense? (Consider what it means for a real neuron to have zero activation in terms of the neuron's resting potential.)
2. Show that the system suppresses noise by requiring that the reflectance values, Θ_i, be greater than some minimum value before they will excite a positive activity in the processing element.

6.1.2 The Instar

The hidden layer of the CPN comprises a set of processing elements known as *instars*. In this section, we shall discuss the instar individually. In the following section, we shall examine the set of instars that operate together to form the CPN hidden layer.

The **instar** is a single processing element that shares its general structure and processing functions with many other processing elements described in this text. We distinguish it by the specific way in which it is trained to respond to input data.

Let's begin with a general processing element, such as the one in Figure 6.4(a). If the arrow representing the output is ignored, the processing element can be redrawn in the starlike configuration of Figure 6.4(b). The inward-pointing arrows identify the instar structure, but we restrict the use of the term *instar* to those units whose processing and learning are governed by the

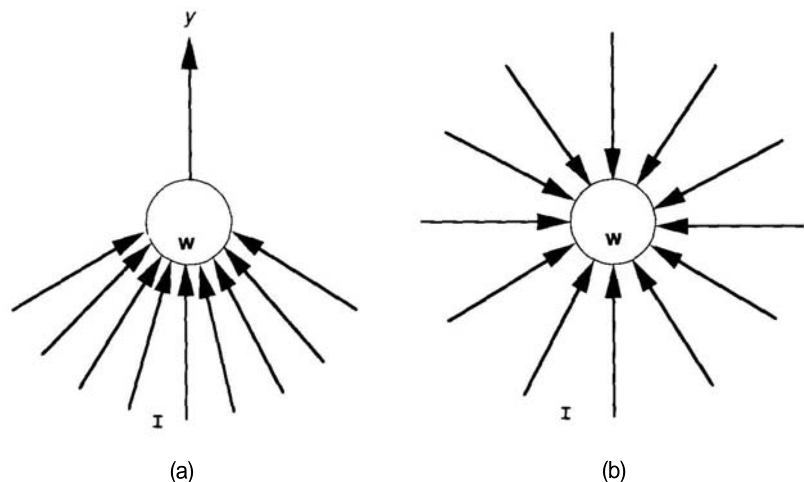


Figure 6.4 This figure shows (a) the general form of the processing element with input vector I, weight vector **w**, and output value y; and (b) the instar form of the processing element in (a). Notice that the arrow representing the output is missing, although it is still presumed to exist.

equations in this section. The net-input value is calculated, as usual, by the dot product of the input and weight vectors, net = I • w. We shall assume that the input vector, I, and the weight vector, w, have been normalized to a length of 1.

The output of the instar is governed by the equation

$$y = -ay + b\,\text{net} \tag{6.4}$$

where a, $b > 0$. The dynamic behavior of y is illustrated in Figure 6.5.

We can solve Eq. (6.4) to get the output as a function of time. Assuming the initial output is zero, and that a nonzero input vector is present from time $t = 0$ until time t,

$$y(t) = \frac{b}{a}\text{net}\left(1 - e^{-at}\right) \tag{6.5}$$

The equilibrium value of $y(t)$ is

$$y^{eq} = \frac{b}{a}\text{net} \tag{6.6}$$

If the input vector is removed at time t', after equilibrium has been reached, then

$$y(t) = y^{eq}e^{-a(t-t')} \tag{6.7}$$

for $t > t'$.

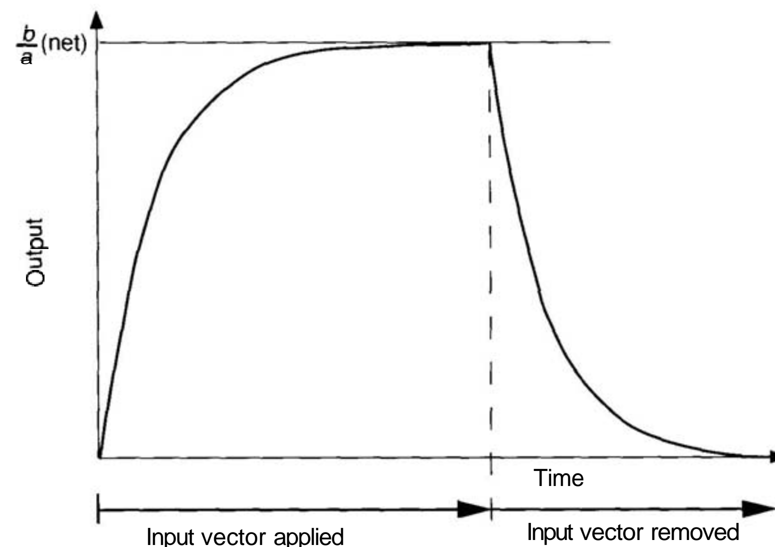


Figure 6.5 This graph illustrates the output response of an instar. When the input vector is nonzero, the output rises to an equilibrium value of (b/a)net. If the input vector is removed, the output falls exponentially with a time constant of $1/a$.

Notice that, for a given a and b, the output at equilibrium will be larger when the net-input value is larger. Figure 6.6 shows a diagram of the weight vector, **w**, of an instar, and an input vector, I. The net-input value determines how close to each other the two vectors are as measured by the angle between them, θ. The largest equilibrium output will occur when the input and weight vectors are perfectly aligned ($\theta = 0$).

If we want the instar to respond maximally to a particular input vector, we must arrange that the weight vector be made identical to the desired input vector. The instar can learn to respond maximally to a particular input vector if the initial weight vector is allowed to change according to the equation

$$\dot{\mathbf{w}} = -c\mathbf{w} + d\mathbf{I}y \tag{6.8}$$

where y is the output of the instar, and $c, d > 0$. Notice the relationship between Eq. (6.8) and the Hebbian learning rule discussed in Chapter 1. The second term on the right side of Eq. (6.8) contains the product of the input and the output of a processing element. Thus, when both are large, the weight on the input connection is reinforced, as predicted by Hebb's theory.

Equation 6.8 is difficult to integrate because y is a complex function of time through Eq. (6.5). We can try to simplify the problem by assuming that y

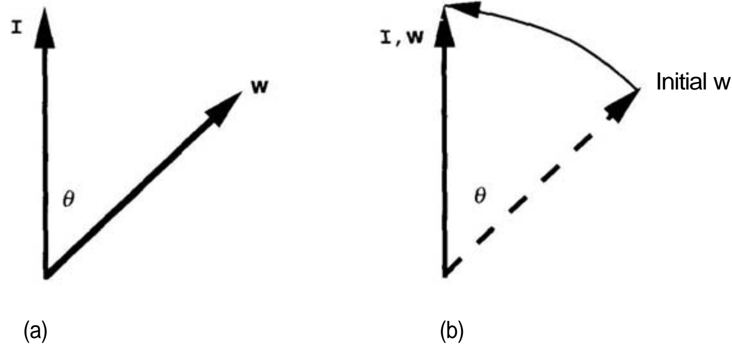


Figure 6.6 This figure shows an example of an input vector and a weight vector on an instar. (a) This figure illustrates the relationship between the input and weight vectors of an instar. Since the vectors are normalized, net $= \mathbf{I} \cdot \mathbf{w} = \|\mathbf{I}\|\|\mathbf{w}\| \cos\theta — \cos\theta$. (b) The instar *learns* an input vector by rotating its weight vector toward the input vector until both vectors are aligned.

reaches its equilibrium value much faster than changes in w can occur. Then, $y = y^{eq} — (a/b)$net. Because net = w • I, Eq. (6.8) becomes

$$\dot{\mathbf{w}} = -c\,\mathbf{w} + d\,\mathbf{I}(\mathbf{w} \cdot \mathbf{I}) \tag{6.9}$$

where we have absorbed the factor a/6 into the constant *d.* Although Eq. (6.9) is still not directly solvable, the assumption that changes to weights occur more slowly than do changes to other parameters is important. We shall see more of the utility of such an assumption in Chapter 8. Figure 6.7 illustrates the solution to Eq. (6.9) for a simple two-dimensional case.

An alternative approach to Eq. (6.8) begins with the observation that, in the absence of an input vector, I, the weight vector will continuously decay away toward zero ($\mathrm{w} = -c\,\mathbf{w}$). This effect can be considered as a forgetfulness on the part of the processing element. To avoid this forgetfulness, we can modify Eq. (6.8) such that any change to the weight vector depends on whether there is an input vector there to be learned. If an input vector is present, then net = w • I will be nonzero. Instead of Eq. (6.8), we can use as the learning law,

$$\mathrm{w} = (-c\mathbf{w} + d\mathbf{I})U(\mathrm{net}) \tag{6.10}$$

where

$$U(\mathbf{net}) = \begin{cases} 1 & \mathrm{net} > 0 \\ 0 & \mathrm{net} = 0 \end{cases}$$

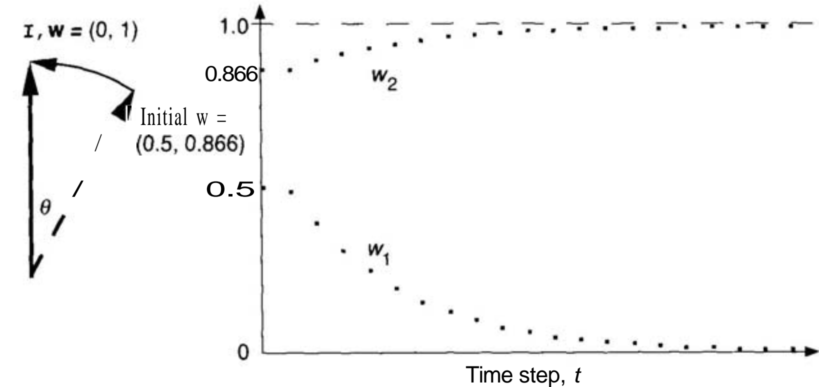


Figure 6.7 Given an input vector $\mathrm{I} = (0, 1)$ and an initial weight vector, $\mathbf{w}(0) = (0.5, 0.866)$, the components, w_1 and w_2, of the weight vector evolve in time according to Eq. (6.9), as shown in the graph. The weight vector eventually aligns itself to the input vector such that $\mathrm{w} = (0, 1) = \mathrm{I}$. For this example, $c = d — 1$.

Equation (6.10) can be integrated directly for $U(\mathrm{net}) = 1$. Notice that $\mathbf{w}^{eq} = (d/c)l$, making $c = d$ a condition that must be satisfied for w to evolve toward an exact match with I. Using this fact, we can rewrite Eq. (6.10) in a form more suitable for later digital simulations:

$$\mathrm{Aw} = \alpha(\mathbf{I} — \mathrm{w}) \tag{6.11}$$

In Eq. (6.11), we have used the approximation $d\mathbf{w}/dt \approx \Delta\mathbf{w}/\Delta t$, and have let $a = c\Delta t$. An approximate solution to Eq. (6.10) would be

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \alpha(\mathbf{I} - \mathbf{w}(t)) \tag{6.12}$$

for $\alpha < 1$; see Figure 6.8.

A single instar learning a single input vector does not provide an interesting case. Let's consider the situation where we have a number of input vectors, all relatively close together in what we shall call a *cluster.* A cluster might represent items of a single class. We would like the instar to learn some form of representative vector of the class: the average, for example. Figure 6.9 illustrates the idea.

Learning takes place in an iterative fashion:

1. Select an input vector, I_i, at random, according to the probability distribution of the cluster. (If the cluster is not uniformly distributed, you should select input vectors more frequently from regions where there is a higher density of vectors.)
2. Calculate $\alpha(\mathbf{I} — \mathrm{w})$, and update the weight vector according to Eq. (6.12).

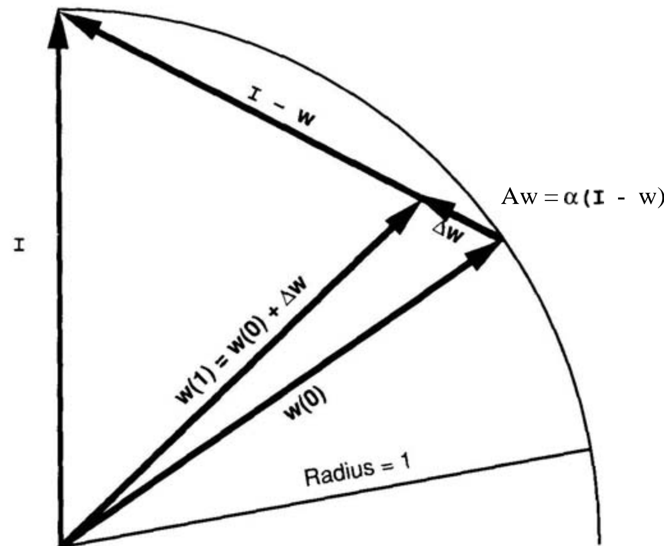


Figure 6.8 The quantity (I - w) is a vector that points from w toward I. In Eq. (6.12), w moves in discrete timesteps toward I. Notice that w does not remain normalized.

3. Repeat steps 1 and 2 for a number of times equal to the number of input vectors in the cluster.
4. Repeat step 3 several times.

The last item in this list is admittedly vague. There is a way to calculate an average error as learning proceeds, which can be used as a criterion for halting the learning process (see Exercise 6.4). In practice, error values are rarely used, since the instar is never used as a stand-alone unit, and other criteria will determine when to stop the training.

It is also a good idea to reduce the value of a as training proceeds. Once the weight vector has reached the *middle* of the cluster, outlying input vectors might pull w out of the area if a is very large.

When the weight vector has reached an average position within the cluster, it should stay generally within a small region around that average position. In other words, the average change in the weight vector, $\langle \Delta \mathbf{w} \rangle$, should become very small. Movements of w in one direction should generally be offset by movements in the opposite direction. If we assume $\langle \Delta \mathbf{w} \rangle = 0$, then Eq. (6.11) shows that

$$\langle \mathbf{w} \rangle = \langle \mathbf{I} \rangle$$

which is what we wanted.

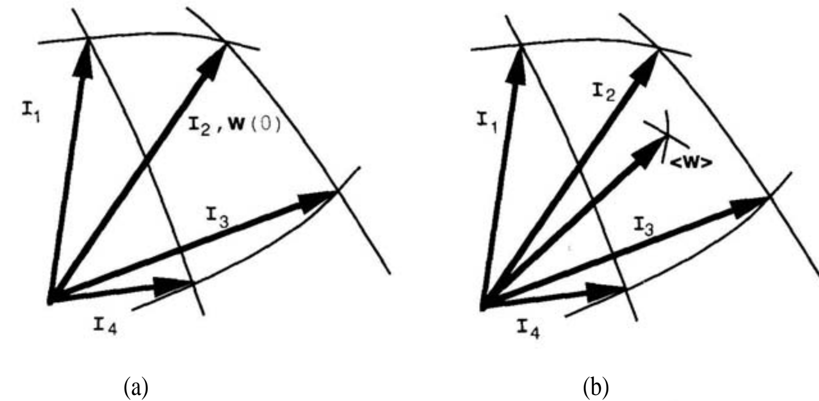


Figure 6.9 This figure illustrates how an instar learns to respond to a cluster of input vectors. (a) To learn a cluster of input vectors, we select the initial weight vector to be equal to some member of the cluster. This initialization ensures that the weight vector is in the right region of space. (b) As learning proceeds, the weight vector will eventually settle down to some small region that represents an average, $\langle \mathbf{w} \rangle$, of all the input vectors.

Now that we have seen how an instar can learn the average of a cluster of input vectors, we can talk about layers of instars. Instars can be grouped together into what is known as a **competitive network.** The competitive network forms the middle layer of the CPN and is the subject of the next section.

Exercise 6.4: For a given input vector, we can define the instar error as the magnitude of the difference between the input vector and the weight vector: $\varepsilon^2 = \|\mathbf{I}_i - \mathbf{w}\|$. Show that the mean squared error can be written as

$$\langle \varepsilon^2 \rangle = 2(1 - \langle \cos \theta_i \rangle)$$

where θ_i is the angle between $\mathbf{I}_i$ and w.

6.1.3 Competitive Networks

In the previous section, we saw how an individual instar could learn to respond to a certain group of input vectors clustered together in a region of space. Suppose we have several instars grouped in a layer, as shown in Figure 6.10, each of which responds maximally to a group of input vectors in a different region of space. We can say that this layer of instars *classifies* any input vector, because the instar with the strongest response for any given input identifies the region of space in which the input vector lies.

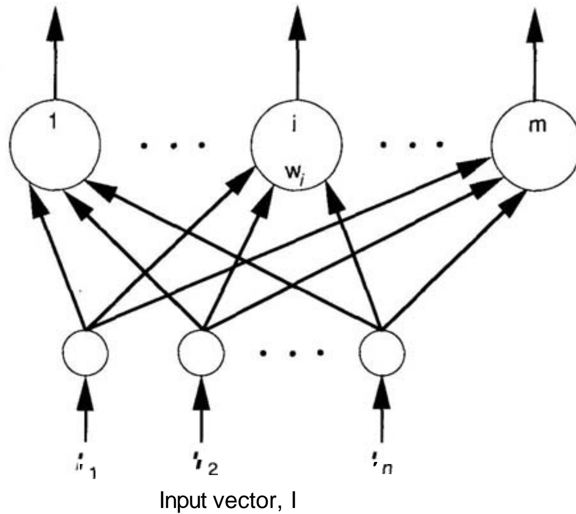


Figure 6.10 A layer of instars arranged in a competitive network. Each unit receives the input vector $\mathbf{I} = (I_1, I_2, \ldots, I_n)^t$ and the ith unit has associated with it the weight vector, $\mathbf{w}_i = (w_{i1}, w_{i2}, \ldots, w_{in})^t$. Net-input values are calculated in the usual way: $\text{net}_i = \mathbf{I} \cdot \mathbf{w}_i$. The winner of the competition is the unit with the largest net input.

Rather than our examining the response of each instar to determine which is the largest, our task would be simpler if the instar with the largest response were the *only* unit to have a nonzero output. This effect can be accomplished if the instars compete with one another for the privilege of turning on. Since there is no external judge to decide which instar has the largest net input, the units must decide among themselves who is the winner. This decision process requires communication among all the units on the layer; it also complicates the analysis, since there are more inputs to each unit than just the input from the previous layer. In the following discussion, we shall be focusing on unit activities, rather than on unit output values.

Figure 6.11 illustrates the interconnections that implement competition among the instars. The unit activations are determined by differential equations. There is a variety of forms that these differential equations can take; one of the simplest is

$$\dot{x}_i = -Ax_i + (B - x_i)[f(x_i) + \text{net}_i] - x_i \left[\sum_{k \neq i} f(x_k) + \sum_{k \neq i} \text{net}_k \right] \qquad (6.13)$$

where A, $B > 0$ and $f(x_i)$ is an output function that we shall specify shortly [2]. This equation should be compared to Eq. (6.2). We can convert Eq. (6.2) to

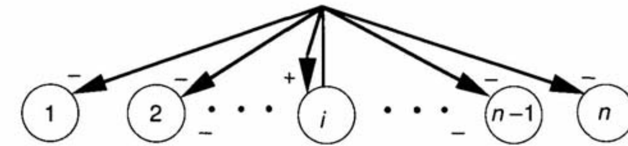


Figure 6.11 An on-center off-surround system for implementing competition among a group of instars. Each unit receives a positive feedback signal to itself and sends an inhibitory signal to all other units on the layer. The unit whose weight vector most closely matches the input vector sends the strongest inhibitory signals to the other units and receives the largest positive feedback from itself.

Eq. (6.13) by replacing every occurrence of Ij in Eq. (6.2) with $f(x_j) + \text{net}_j$, for all j. The relationship between the constants A and B, and the form of the function, $f(xj)$, determine how the solutions to Eq. (6.13) evolve in time. We shall now look at specific cases.

Equation (6.13) is somewhat easier to analyze if we convert it to a pair of equations: one that describes the reflectance variables, $X_i - x_i / \sum_k x_k$, and one that describes the total pattern intensity, $x = \sum_k x_k$. First, rearrange Eq. (6.13) as follows:

$$\dot{x}_i = -Ax_i + B[f(x_i) + \text{net}_i] - x_i \left[\sum_k f(x_k) + \sum_k \text{net}_k \right]$$

Next, sum over i to get

$$x = -Ax + (B - x) \left[\sum_k f(xX_k) + \sum_k \text{net}_k \right] \qquad \textbf{(6.14)}$$

Now substitute xXi into Eq. (6.13) and use Eq. (6.14) to simplify the result. If we make the definition $g(w) = w^{-1} f(w)$, then we get

$$xX_i = BxX_i \sum_k X_k [g(xX_i) - g(xX_k)] + B(1 - X_i)\text{net}_i - BX_i \sum_{k \neq i} \text{net}_k \qquad (6.15)$$

We can now evaluate the asymptotic behavior of Eqs. (6.14) and (6.15). Let's begin with the simple, linear case of $f(w) = w$. Since $g(w) = w^{-1} f(w)$, $g(w) = 1$ and the first term on the right of Eq. (6.15) is zero. The reflectance variables stabilize at

$$X_i = \frac{\text{net}_i}{\sum_k \text{net}_k}$$

so the activities become

$$x_i = x^{eq} \frac{\text{net}_i}{\sum_k \text{net}_k}$$

where x^{eq} comes from setting Eq. (6.14) equal to zero. This equation shows that the units will accurately register any pattern presented as an input. Now let's look at what happens after the input pattern is removed.

Let net, = 0 for all *i*. Then, $\dot{X}_i$ is identically zero for all time and the reflectance variables remain constant. The unit activations now depend only on *x*, since *x*, — xX_i. Equation (6.14) reduces to

$$x = (B - A - x)x$$

If $B < A$, then $x < 0$ and $x \to 0$. However, if $B > A$, then $x \to B - A$ and the activity pattern becomes stored permanently on the units. This behavior is unlike the behavior of the input units described by Eq. (6.2), where removal of the input pattern always resulted in a decay of the activities. We shall call this storage effect **short-term memory** (STM). Even though the effect appears to be permanent, we can assume the existence of a reset mechanism that will remove the current pattern so that another can be stored. Figure 6.12 illustrates a simple example for the linear output function.

For our second example, we assume a faster-than-linear output function, $f(w) = w^2$. In this case $g(w) = w$. Notice that the first term on the right of Eq. (6.15) contains the factor $[g(xX_i) - g(xX_k)]$. For the quadratic output function, this expression reduces to $x[X_i - X_k]$. If $X_i > X_k$ for all values of $k \neq$?', then the first term in Eq. (6.14) is an excitatory term. If *Xi* $< X_k$ for A- $\neq$?', then the first term in Eq. (6.14) becomes an inhibitory term. Thus, this network tends to enhance the activity of the unit with the largest value of *Xi*. This effect is illustrated in Figure 6.13. After the input pattern is removed, $\dot{X}_i$ will be greater than zero for only the unit with the largest value of *Xi*.

Exercise *6.5:* Use Eq. (6.14) to show that, after the input pattern has been removed, the total activity of the network, *x*, is bounded by the value of *B*.

We now have an output function that can be used to implement a winner-take-all competitive network. The quadratic output function (or, for that matter, any function $f(w) = w^n$, where $n > 1$) can be used in off-surround, inhibitory connections to suppress all inputs except the largest. This effect is the ultimate in noise suppression: The network assumes that everything except the largest signal is noise.

It is possible to combine the qualities of noise suppression with the ability to store an accurate representation of an input vector: Simply arrange for the output function to be faster than linear for small activities, and linear for larger activities. If we add the additional constraint that the unit output must be bounded at all times, we must have the output function increase at a less-than-linear rate for large values of activity. This combination results in a sigmoid output function, as illustrated in Figure 6.14.

The mathematical analysis of Eqs. (6.14) and (6.15) with a sigmoid output function is considerably more complicated than it was for the other twocases. All the cases considered here, as well as many others, are treated in depth by

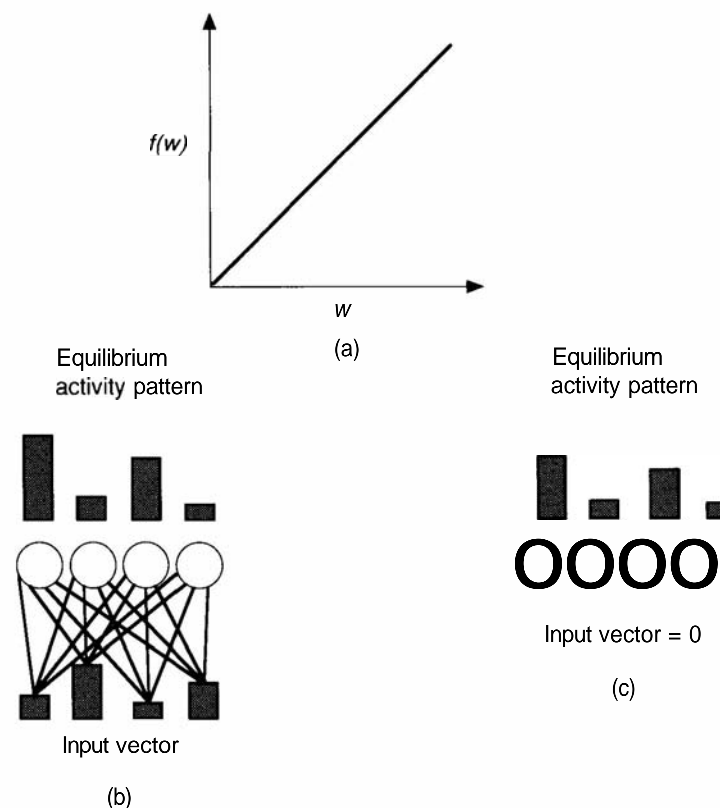


Figure 6.12 This series of figures shows the result of applying a certain input vector to units having a linear output function. (a) The graph of the output function, $f(w) = w$. (b) This figure shows the result of the sustained application of the input vector. The units reach equilibrium activities as shown. (c) After removal of the input vector, the units reach an equilibrium such that the pattern is stored in STM.

Grossberg [4]. Use of the sigmoid results in the existence of a **quenching threshold** (QT). Units whose net inputs are above the QT will have their activities enhanced. The effect is one of **contrast enhancement.** An extreme example is illustrated in Figure 6.14.

Reference back to Figure 6.1 or 6.2 will reveal that there are no obvious interconnections among the units on the competitive middle layer. In a digital simulation of a competitive network, the actual interconnections are unnecessary. The CPU can act as an external judge to determine which unit has the largest net-

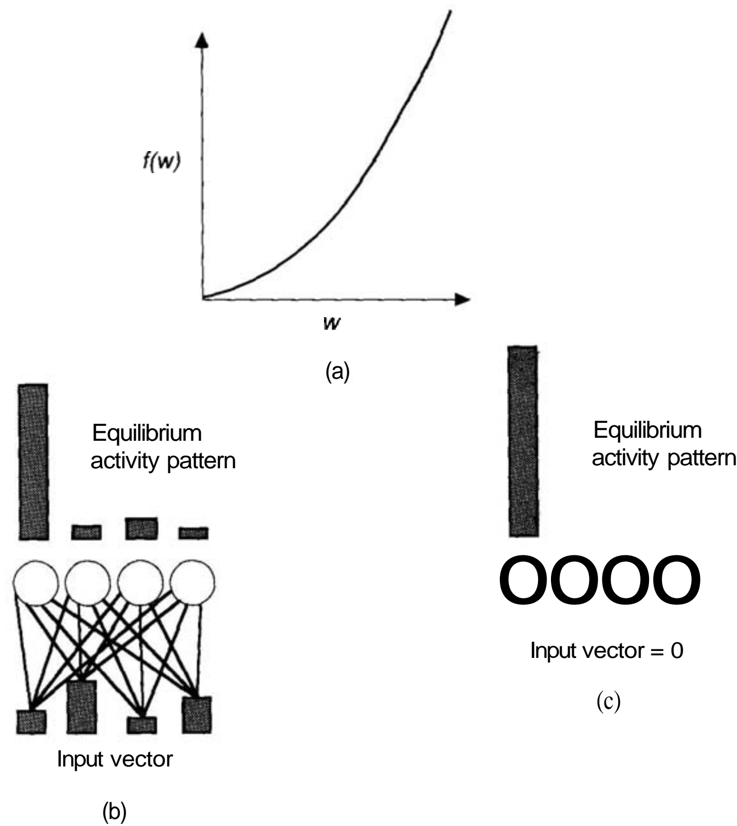


Figure 6.13 This series of figures is analogous to those in Figure 6.12, but with units having a quadratic output function. (a) The graph of the quadratic output function. (b) While the input vector is present, the network tends to enhance the activity of the unit with the largest activity. For the given input pattern, the unit activities reach the equilibrium values shown. (c) After the input pattern is removed, all activities but the largest decay to zero.

input value. The winning unit would then be assigned an output value of 1. The situation is similar for the input layer. In a software simulation, we do not need on-center off-surround interactions to normalize the input vector; that can also be done easily by the CPU. These considerations aside, attention to the underlying theory is essential to understanding. When digital simulations give way to neural-network integrated circuitry, such an understanding will be required.

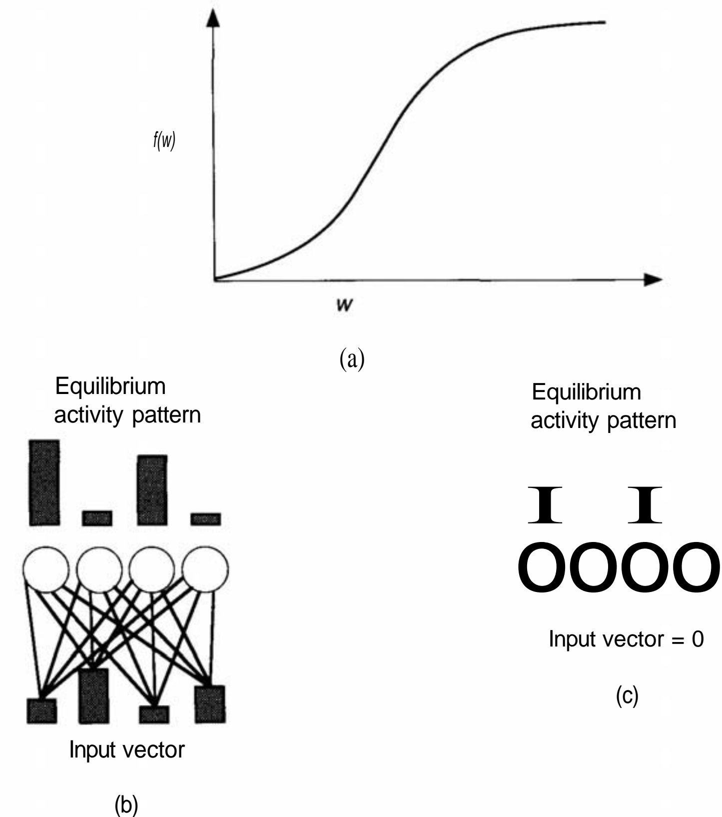


Figure 6.14 This figure is analogous to Figures 6.12 and 6.13, but with units having a sigmoid output function. (a) The sigmoid output function combines noise suppression at low activities, linear pattern storage at intermediate values, and a bounded output at large activity values. (b) When the input vector is present, the unit activities reach an equilibrium value, as shown. (c) After removal of the input vector, the activities above a certain threshold are enhanced, whereas those below the threshold are suppressed.

6.1.4 The Outstar

The final leg of our journey through CPN components brings us to Grossberg's outstar structure. As Figure 6.15 shows, an outstar is composed of all of the units in the CPN outer layer and a single hidden-layer unit. Thus, the outer-layer units participate in several different outstars: one for each unit in the middle layer.

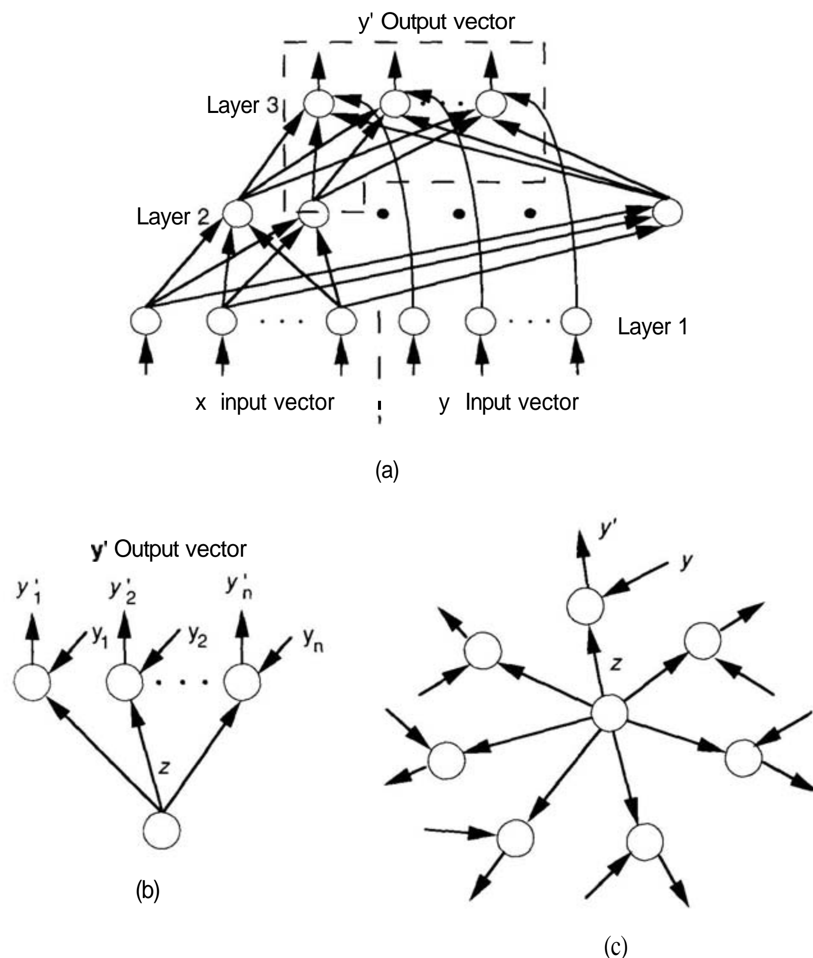


Figure 6.15 This figure illustrates the *outstar* and its relationship to the CPN architecture. (a) The dotted line encompasses one of the outstar structures in the CPN. The line is intentionally drawn through the middle-layer unit to indicate the dual functionality of that unit. Each middle-layer unit combines with the outer layer to form an individual outstar. (b) A single outstar unit is shown. The output units of the outstar have two inputs: *z*, from the connecting unit of the previous layer, and y_i, which is the training input. The training input is present during only the learning period. The output of the outstar is the vector $\mathbf{y}' = (y_1', y_2', \ldots, y_n')^t$. *(c)* The outstar is redrawn in a suggestive configuration. Note that the arrows point outward in a configuration that is complementary to the instar.

In Chapter 1, we gave a brief description of Pavlovian conditioning in terms of Hebbian learning. Grossberg argues that the outstar is the minimal neural architecture capable of classical conditioning [3]. Consider the outstar shown in Figure 6.16. Initially, the **conditioned stimulus** (CS) (e.g., a ringing bell) is assumed to be unable to elicit a response from any of the units to which it is connected. An **unconditioned stimulus** (UCS) (the sight of food) can cause an **unconditioned response** (UCR) (salivation). If the CS is present while the UCS is causing the UCR, then the strength of the connection from the CS unit to the UCR unit will also be increased, in keeping with Hebb's theory (see Chapter 1). Later, the CS will be able to cause a **conditioned response** (CR) (the same as the UCR), even if the UCS is absent.

The behavior of the outstars in the CPN resembles this classical conditioning. During the training period, the winner of the competition on the hidden layer turns on, providing a single CS to the output units. The UCS is supplied by the y-vector portion of the input layer. Because we want the network to learn the actual y vector, the UCR will be the same as the y vector, within a constant multiplicative factor. After training is complete, the appearance of the CS will cause the CR value (the $\mathbf{y}'$ vector) to appear at the output units, even though the UCS values will be all zero.

In the CPN, the hidden layer participates in both the instar and outstar structures of the network. The function of the competitive instars is to recognize an input pattern through a winner-take-all competition. Once a winner has been declared, that unit becomes the CS for an outstar. The outstar associates some value or identity with the input pattern. The instar and outstar complement each other in this fashion: The instar recognizes an input pattern and classifies it; the outstar identifies or names the selected class. This behavior led one researcher to note that the instar is dumb, whereas the outstar is blind [9].

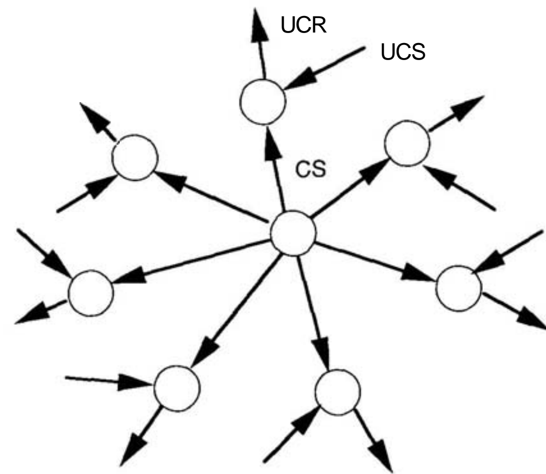
The equations that govern the processing and learning of the outstar are similar in form to those for the instar. During the training process, the output values of the outstar can be calculated from

$$y_i = -ay_i' + by_i + c\,\mathrm{net}_i$$

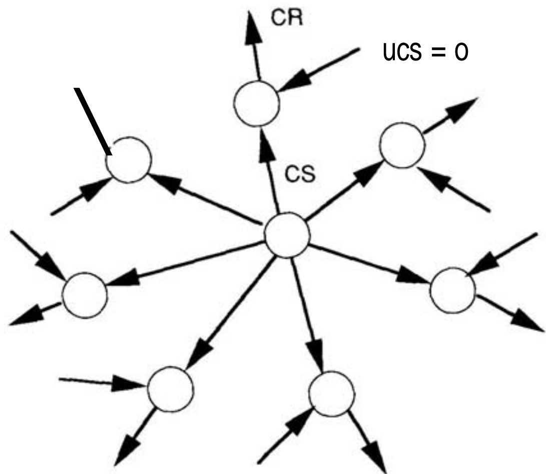
which is similar to Eq. (6.4) for the instar except for the additional term due to the training input, y_i. The parameters *a, b,* and *c,* are all assumed to be positive. The value of net_i is calculated in the usual way as the sum of products of weights and input values from the connecting units. For the outstar and the CPN, only a single connecting unit has a nonzero output at any given time. Even though each output unit of the outstar has an entire weight vector associated with it, the net input reduces to a single term, $w_{ij}z$, where z is the output of the connecting unit. In the case of the CPN, $z = 1$. Therefore, we can write

$$\dot{y}_i' = -ay_i' + byi + cw_{ij} \qquad \textbf{(6.16)}$$

where the jth unit on the hidden layer is the connecting unit. In its most general form, the parameters a, 6, and c in Eq. (6.16) are functions of time. Here, we



(a)



(b)

Figure 6.16 This figure shows an implied classical-conditioning scenario. (a) During the conditioning period, the CS and the UCS excite one of the output units simultaneously. (b) After conditioning, the presence of the CS alone can excite the CR without exciting any of the other output units.

shall consider them to be constants for simplicity. For the remainder of this discussion, we shall drop the j subscript from the weight.

After training is complete, no further changes in w_i take place and the training inputs, y_i, are absent. Then, the output of the outstar is

$$\dot{y}'_i = -ay'_i + cw_i^{eq} \tag{6.17}$$

where w_i^{eq} is the fixed weight value found during training.

The weights evolve according to an equation almost identical to Eq. (6.10) for the instar:

$$w_i = (-dw_i + ey_i z)U(z) \tag{6.18}$$

Notice that the second term in Eq. (6.18) contains the training input, y_i, not the unit output, y'_i. The $U(z)$function ensures that no unlearning takes place when there is no training input present, or when the connecting unit is off $(z — 0)$. Since both z and $U(z)$ are 1 for the middle-layer unit that wins the competition, Eq. (6.18) becomes

$$Wi = -dwi + ey_i \tag{6.19}$$

for the connections from the winning, middle-layer unit. Connections from other middle-layer units do not participate in the learning.

Recall that a given instar can learn to recognize a cluster of input vectors. If the desired CPN outputs (the y_is) corresponding to each vector in the cluster are all identical, then the weights eventually reach the equilibrium values:

$$w_i^{eq} = \frac{e}{d} y_i$$

If, on the other hand, each input vector in the cluster has a slightly different output value associated with it, then the outstar will learn the average of all of the associated output values:

$$w_i^{eq} = \frac{e}{d} \langle y_i \rangle$$

Using the latter value for the equilibrium weight, Eq. (6.17) shows that the output after training reaches a steady state value of

$$y_i'^{eq} = \frac{c}{a}\frac{e}{d} \langle y_i \rangle.$$

Since we presumably want $y_i'^{eq} = \langle y_i \rangle$, we can require that $a = c$ in Eq. (6.17) and that $d = e$ in Eq. (6.18). Then,

$$y_i'^{eq} = \langle y_i \rangle = w_i^{eq} \tag{6.20}$$

For the purpose of digital simulation, we can approximate the solution to Eq. (6.19) by

$$w_i(t+1) = w_i(t) + \beta(y_i - w_i(t)) \tag{6.21}$$

following the same procedure that led to Eq. (6.12) for the instar.

6.2 CPN DATA PROCESSING

We are now in a position to combine the component structures from the previous section into the complete CPN. We shall still consider only the forward-mapping CPN for the moment. Moreover, we shall assume that we are performing a digital simulation, so it will not be necessary to model explicitly the interconnects for the input layer or the competitive layer.

6.2.1 Forward Mapping

Assume that all training has occurred and that the network is now in a production mode. We have an input vector, I, and we would like to find the corresponding y vector. The processing is depicted in Figure 6.17 and proceeds according to the following algorithm:

1. Normalize the input vector, $x_i = I_i/(\sqrt{\sum_n I_i^2})$.
2. Apply the input vector to the x-vector portion of layer 1. Apply a zero vector to the **y**-vector portion of layer 1.
3. Since the input vector is already normalized, the input layer only distributes it to the units on layer 2.
4. Layer 2 is a winner-take-all competitive layer. The unit whose weight vector most closely matches the input vector wins and has an output value of 1. All other units have outputs of 0. The output of each unit can be calculated according to

$$z_i = \begin{cases} 1 & \|\text{net}_i\| > \|\text{net}_j\| \text{ for all } j \neq i \\ 0 & \text{otherwise} \end{cases} \tag{6.22}$$

5. The single winner on layer 2 excites an outstar.

Each unit in the outstar quickly reaches an equilibrium value equal to the value of the weight on the connection from the winning layer 2 unit [see Eq. (6.20)]. If the ith unit wins on the middle layer, then the output layer produces an output vector $\mathbf{y}' = (w_{1i}, w_{2i}, \ldots, w_{mi})^t$, where m represents the number of units on the output layer. A simple way to view this processing is to realize that the equilibrium output of the outstar is equal to the outstar's net input,

$$y_k'^{eq} = \sum_j w_{kj} z_j \tag{6.23}$$

Since $z_j = 0$ unless $j = i$, then $y_k'^{eq} = w_{ki} z_i = w_{ki}$, which is consistent with the results obtained in Section 6.1.

This simple algorithm uses equilibrium, or asymptotic, values of node activities and outputs. We thus avoid the need to solve numerically all the corresponding differential equations.

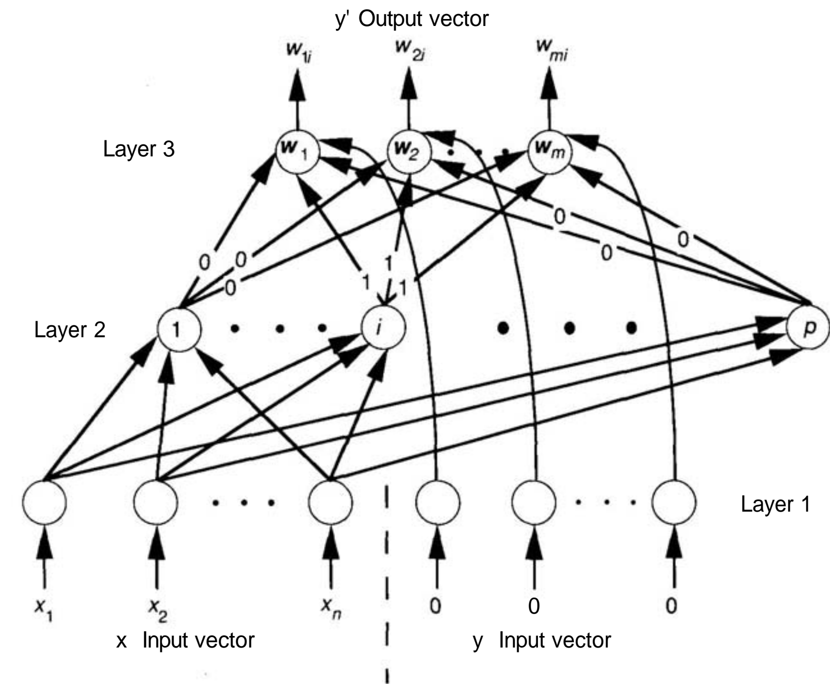


Figure 6.17 This figure shows a summary of the processing done on an input vector by the CPN. The input vector, $(x_1, x_2, \ldots, x_n)^t$ is distributed to all units on the competitive layer. The ith unit wins the competition and has an output of 1; all other competitive units have an output of 0. This competition effectively selects the proper output vector by exciting a single connection to each of the outstar units on the output layer.

6.2.2 Training the CPN

Here again, we assume that we are performing a digital simulation of the CPN. Although this assumption does not eliminate the need to find numerical solutions to the differential equations, we can still take advantage of prenormalized input vectors and an external judge to determine winners on the competitive layer. We shall also assume that a set of training vectors has been defined adequately. We shall have more to say on that subject in a later section.

Because there are two different learning algorithms in use in the CPN, we shall look at each one independently. In fact, it is a good idea to train the competitive layer completely before beginning to train the output layer.

The competitive-layer units train according to the instar learning algorithm described in Section 6.1. Since there will typically be many instars on the competitive layer, the iterative training process described earlier must be amended slightly. Here, as in Section 6.1, we assume that a cluster of input vectors forms a single class. Now, however, we have the situation where we may have several clusters of vectors, each cluster representing a different class. Our learning procedure must be such that each instar learns (wins the competition) for all the vectors in a single cluster. To accomplish the correct classification for each class of input vectors, we must proceed as follows:

1. Select an input vector from among all the input vectors to be used for training. The selection should be random according to the probability distribution of the vectors.
2. Normalize the input vector and apply it to the CPN competitive layer.
3. Determine which unit wins the competition by calculating the net-input value for each unit and selecting the unit with the largest (the unit whose weight vector is closest to the input vector in an inner-product sense).
4. Calculate $\alpha(\mathbf{x} - \mathbf{w})$ for the winning unit only, and update that unit's weight vector according to Eq. (6.12):

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \alpha(\mathbf{x} - \mathbf{w})$$

5. Repeat steps 1 through 4 until all input vectors have been processed once.
6. Repeat step 5 until all input vectors have been classified properly. When this situation exists, one instar unit will win the competition for all input vectors in a certain cluster. Note that there might be more that one cluster corresponding to a single class of input vectors.
7. Test the effectiveness of the training by applying input vectors from the various classes that were *not* used during the training process itself. If any misclassifications occur, additional training passes through step 6 may be required, even though all the training vectors are being classified correctly. If training ends too abruptly, the **win region** of a particular unit may be offset too much from the center of the cluster, and outlying vectors may be misclassified. We define an instar's *win region* as the region of vector space containing vectors for which that particular instar will win the competition. (See Figure 6.18.)

An issue that we have overlooked in our discussion is the question of initialization of the weight vectors. For all but the simplest problems, random initial weight vectors will not be adequate. We already hinted at an initialization method earlier: Set each weight vector equal to a representative of one of the clusters. We shall have more to say on this issue in the next section.

Once satisfactory results have been obtained on the competitive layer, training of the outstar layer can occur. There are several ways to proceed based on the nature of the problem.

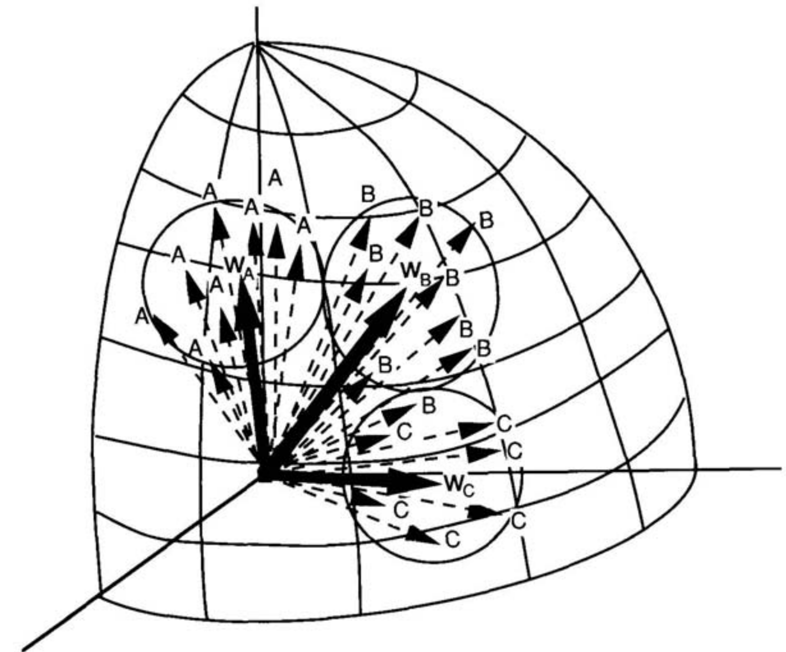


Figure 6.18 In this drawing, three clusters of vectors represent three distinct classes: A, B, and C. Normalized, these vectors end on the unit hypersphere. After training, the weight vectors on the competitive layer have settled near the centroid of each cluster. Each weight vector has a *win region* represented, although not accurately, by the circles drawn on the surface of the sphere around each cluster. Note that one of the B vectors encroaches into C's win region indicating that erroneous classification is possible in some cases.

Suppose that each cluster of input vectors represents a class, and all of the vectors in a cluster map to the identical output vector. In this case, no iterative training algorithm is necessary. We need only to determine which hidden unit wins for a particular class. Then, we simply assign the weight vector on the appropriate connections to the output layer to be equal to the desired output vector. That is, if the ith hidden unit wins for all input vectors of the class for which A is the desired output vector, then we set $w_{ki} = A_k$, where w_{ki} is the weight on the connection from the *ith* hidden unit to the kth output unit.

If each input vector in a cluster maps to a different output vector, then the outstar learning procedure will enable the outstar to reproduce the average of

those output vectors when any member of the class is presented to the inputs of the CPN. If the average output vector for each class is known or can be calculated in advance, then a simple assignment can be made as in the previous paragraph: Let $w_{ki} = \langle A_k \rangle$.

If the average of the output vectors is not known, then an iterative procedure can be used based on Eq. (6.21).

1. Apply a normalized input vector, $\mathbf{x}_k$, and its corresponding output vector, $\mathbf{y}_k$, to the x and y inputs of the CPN, respectively.
2. Determine the winning competitive-layer unit.
3. Update the weights on the connections from the winning competitive unit to the output units according to Eq. (6.21):

$$w_i(t+1) = w_i(t) + \beta(y_{ki} - w_i(t))$$

4. Repeat steps 1 through 3 until all vectors of all classes map to satisfactory outputs.

6.2.3 Practical Considerations

In this section, we shall examine several aspects of CPN design and operation that will influence the results obtained using this network. The CPN is deceptively simple in its operation and there are several pitfalls. Most of these pitfalls can be avoided through a careful analysis of the problem being solved before an attempt is made to model the problem with the CPN. We cannot cover all eventualities in this section. Instead, we shall attempt to illustrate the possibilities in order to raise your awareness of the need for careful analysis.

The first consideration is actually a combination of two: the number of hidden units required, and the number of exemplars, or training vectors, needed for each class. It stands to reason that there must be at least as many hidden nodes as there are classes to be learned. We have been assuming that each class of input vectors can be identified with a cluster of vectors. It is possible, however, that two completely disjoint regions of space contain vectors of the same class. In such a situation, more than one competitive node would be required to identify the input vectors of a single class. Unfortunately, for problems with large dimensions, it may not always be possible to determine that such is the case in advance. This possibility is one reason why more than one representative for each class should be used during training, and also why the training should be verified with other representative input vectors.

Suppose that a misclassification of a test vector does occur after all of the training vectors are classified correctly. There are several possible reasons for this error. One possibility is that the set of exemplars did not adequately represent the class, so the hidden-layer weight vector did not find the true centroid. Equivalently, training may not have continued for a sufficient time to center the weight vector properly; this situation is illustrated in Figure 6.19.

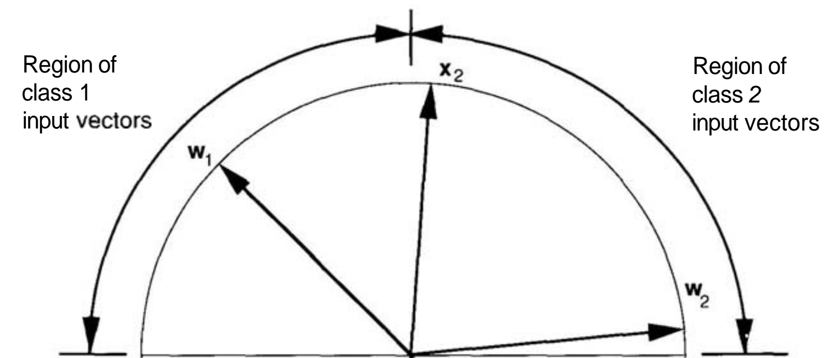


Figure 6.19 In this example, weight vector $\mathbf{w}_1$ learns class 1 and $\mathbf{w}_2$ learns class 2. The input vectors of each class extend over the regions shown. Since $\mathbf{w}_2$ has not learned the true centroid of class 2, an outlying vector, $\mathbf{x}_2$, is actually closer to $\mathbf{w}_1$ and is classified erroneously as *a* member of class 1.

One solution to these situations is to add more units on the competitive layer. Caution must be used, however, since the problem may be exacerbated. A unit added whose weight vector appears at the intersection between two classes may cause misclassification of many input vectors of the original two classes. If a threshold condition is added to the competitive units, a greater amount of control exists over the partitioning of the space into classes. A threshold prevents a unit from winning if the input vector is not within a certain minimum angle, which may be different for each unit. Such a condition has the effect of limiting the size of the win region of each unit.

There are also problems that can occur during the training period itself. For example, if the distribution of the vectors of each class changes with time, then competitive units that were coded originally for one class may get recoded to represent another. Moreover, after training, moving distributions will result in serious classification errors. Another situation is illustrated in Figure 6.20. The problem there manifests itself in the form of a **stuck vector;** that is, one unit that never seems to win the competition for any input vector.

The stuck-vector problem leads us to an issue that we touched on earlier: the initialization of the competitive-unit weight vectors. We stated in the previous section that a good strategy for initialization is to assign each weight vector to be identical to one of the prototype vectors for each class. The primary motivation for using this strategy is to avoid the stuck-vector problem.

The extreme case of the stuck-vector problem can occur if the weight vectors are initialized to random values. Training with weight vectors initialized in this manner could result in *all but one* of the weight vectors becoming stuck. A

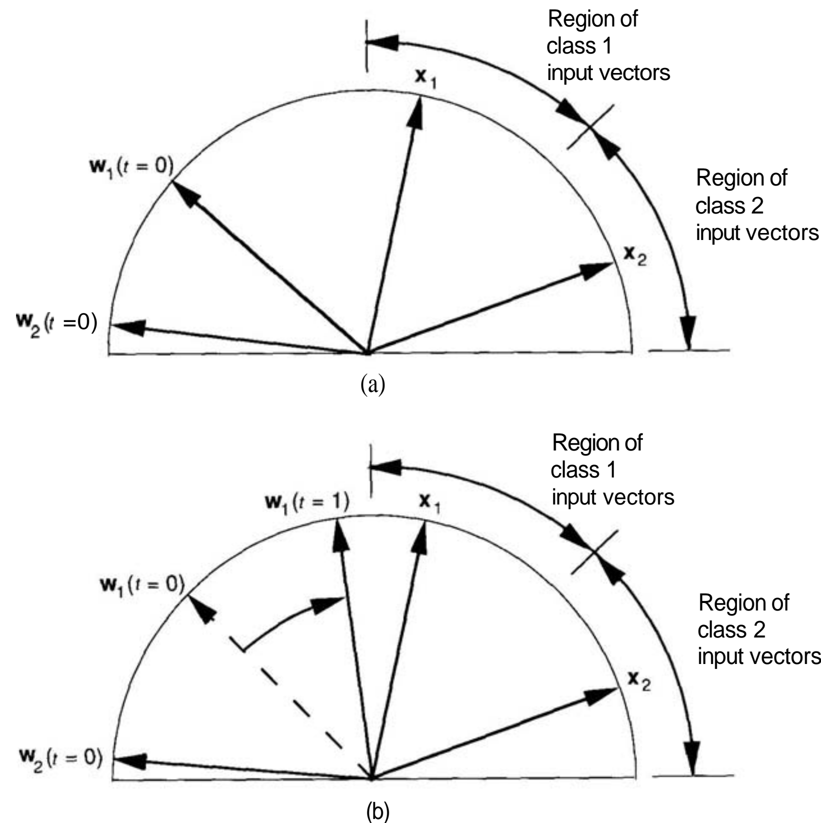


Figure 6.20 This figure illustrates the *stuck-vector problem.* (a) In this example, we would like $\mathbf{w}_1$ to learn the class represented by $\mathbf{x}_1$, and $\mathbf{w}_2$ to learn $\mathbf{x}_2$. (b) Initial training with $\mathbf{x}_1$ has brought $\mathbf{w}_1$ closer to $\mathbf{x}_2$ than $\mathbf{w}_2$ is. Thus, $\mathbf{w}_1$ will win for either $\mathbf{x}_1$ or $\mathbf{x}_2$, and $\mathbf{w}_2$ will never win.

single weight vector would win for every input vector, and the network would not learn to distinguish between any of the classes on input vectors.

This rather peculiar occurrence arises due to a combination of two factors: (1) in a high-dimensional space, random vectors are all nearly orthogonal to one another (their dot products are near 0), and (2) it is not unlikely that all input vectors for a particular problem are clustered within a single region of space. If these conditions prevail, then it is possible that only one of the random weight vectors lies within the same region as the input vectors. Any input vector would have a large dot product with that one weight vector only, since all other weight vectors would be in orthogonal regions.

Another approach to dealing with a stuck vector is to endow the competitive units with a **conscience.** Suppose that the probability that a particular unit wins the competition was inversely proportional to the number of times that unit won in the past. If a unit wins too often, it simply shuts down, allowing others to win for a change. Incorporating this feature can *unstick* a stuck vector resulting from a situation such as the one shown in Figure 6.20.

In contrast to the competitive layer, the layer of outstars on the output layer has few potential problems. Weight vectors can be randomized initially, or set equal to 0 or to some other convenient value. In fact, the only real concern is the value of the parameter, *(3,* in the learning law, Eq. (6.21). Since Eq. (6.21) is a numerical approximation to the solution of a differential equation, *0* should be kept suitably small, (0 < *(3* ≪ 1), to keep the solution well-behaved. As learning proceeds, β can be increased somewhat as the difference term, *(yi* — $w_i(t)$), becomes smaller.

The parameter *a* in the competitive-layer learning law can start out somewhat larger than *(3. A* larger initial *a* will bring weight vectors into alignment with exemplars more quickly. After a few passes, *a* should be *reduced* rather than increased. A smaller *a* will prevent outlying input vectors from pulling the weight vector very far from the centroid region.

A final caveat concerns the types of problems suitable for the CPN. We stated at the beginning of the chapter that the CPN is useful in many situations where other networks, especially backpropagation, are also useful. There is, however, one class of problems that can be solved readily by the BPN that cannot be solved at all by the CPN. This class is characterized by the need to perform a generalization on the input vectors in order to discover certain features of the input vectors that correlate to certain output values. The parity problem discussed in the next paragraph illustrates the point.

A backpropagation network with an input vector having, say, eight bits can learn easily to distinguish between vectors that have an even or odd number of 1s. A BPN with eight input units, eight hidden units, and one output unit suffices to solve the problem [10]. Using a representative sample of the 256 possible input vectors as a training set, the network learns essentially to count the number of 1s in the input vector. This problem is particularly difficult for the CPN because the network must separate vectors that differ by only a single bit. If your problem requires this kind of generalization, use a BPN.

6.2.4 The Complete CPN

Our discussion to this point has focused on the forward-mapping CPN. We wish to revisit the complete, forward- and reverse-mapping CPN described in the introduction to this chapter. In Figure 6.21, the full CPN (see Figure 6.1) is redrawn in a manner similar to Figure 6.2. Describing in detail the processing done by the full CPN would be largely repetitive. Therefore, we present a summary of the equations that govern the processing and learning.

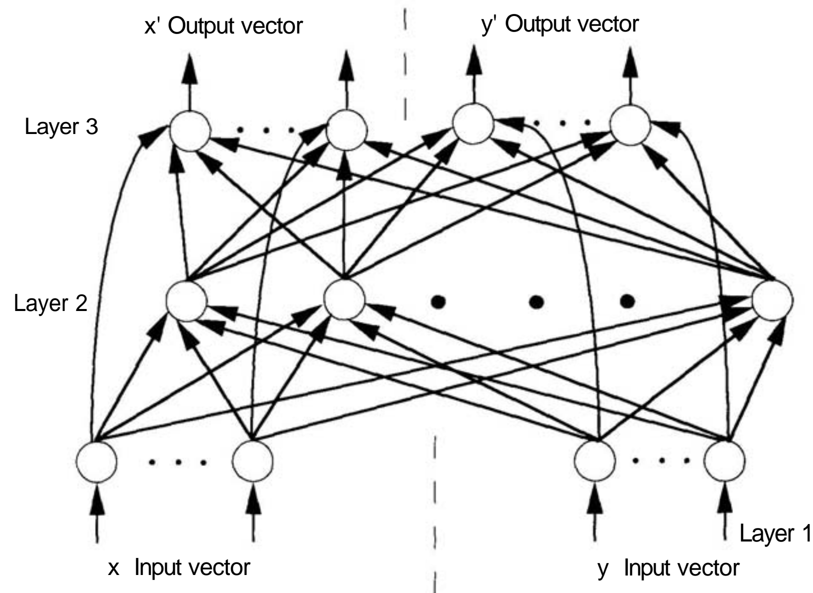


Figure 6.21 The full CPN architecture is redrawn from Figure 6.1. Both x and y input vectors are fully connected to the competitive layer. The x inputs are connected to the $\mathbf{x}'$ output units, and the y inputs are connected to the $\mathbf{y}'$ outputs.

Both x and y input vectors must be normalized for the full CPN. As in the forward-mapping CPN, both x and y are applied to the input units during the training process. After training, inputs of (x, 0) will result in an output of $\mathbf{y}' = \Phi(\mathbf{x})$, and an input of $(\emptyset, \mathbf{y})$ will result in an output of $\mathbf{x}'$.

Because both x and y vectors are connected to the hidden layer, there are two weight vectors associated with each unit. One weight vector, r, is on the connections from the x inputs; another weight vector, s, is on the connections from the y inputs.

Each unit on the competitive layer calculates its net input according to

$$\text{net}_i = \mathbf{r} \cdot \mathbf{x} + \mathbf{s} \cdot \mathbf{y}$$

The output of the competitive layer units is

$$z_i = \begin{cases} 1 & \text{net}_i = \max\{\text{net}_k\} \\ 0 & \text{otherwise} \end{cases}$$

During the training process

$$\mathbf{r}_i = \alpha_x(\mathbf{x} - \mathbf{r}_i)z_i$$
$$\mathbf{s}_i = \alpha_y(\mathbf{y} - \mathbf{s}_i)z_i$$

As with the forward-mapping network, only the winning unit is allowed to learn for a given input vector.

Like the input layer, the output layer is split into two distinct parts. The y' units have weight vectors $\mathbf{w}_i$, and the x' units have weight vectors $\mathbf{v}_i$. The learning laws are

$$\dot{w}_{ij} = (-cw_{ij} + dy_i)z_j$$

and

$$\dot{v}_{ij} = (-ev_{ij} + fx_i)z_j$$

Once again, only weights for which $Zj \neq 0$ are allowed to learn.

Exercise 6.6: What will be the result, after training, of an input of $(\mathbf{x}_a, \mathbf{y}_b)$, where $\mathbf{x}_a = \Phi^{-1}(\mathbf{y}_a)$ and $\mathbf{y}_b = \Phi(\mathbf{x}_b)$?

6.3 AN IMAGE-CLASSIFICATION EXAMPLE

In this section, we shall look at an example of how the CPN can be used to classify images into categories. In addition, we shall see how a simple modification of the CPN will allow the network to perform some interpolation at the output layer.

The problem is to determine the angle of rotation of the principal axis of an object in two dimensions, directly from the raw video image of the object [1]. In this case, the object is a model of the Space Shuttle that can be rotated 360 degrees about a single axis of rotation. Numerical algorithms as well as pattern-matching techniques exist that will solve this problem. The neural-network solution possesses some interesting advantages, however, that may recommend it over these traditional approaches.

Figure 6.22 shows a diagram of the system architecture for the spacecraft orientation system. The video camera, television monitor, and robot all interface to a desktop computer that simulates the neural network and houses a video frame-grabber board. The architecture is an example of how a neural network can be embedded as a part of an overall system.

The system uses a CPN having 1026 input units (1024 for the image and 2 for the training inputs), 12 hidden units, and 2 output units. The units on the middle layer learn to divide the input vectors into different classes. There are 12 units in this layer, and 12 different input vectors are used to train the network. These 12 vectors represent images of the shuttle at 30-degree increments (0°, 30°, . . . , 330°). Since there are 12 categories and 12 training vectors, training of the competitive layer consists of setting each unit's weight equal to one of the (normalized) input vectors. The output layer units learn to associate the correct sine and cosine values with each of the classes represented on the middle layer.

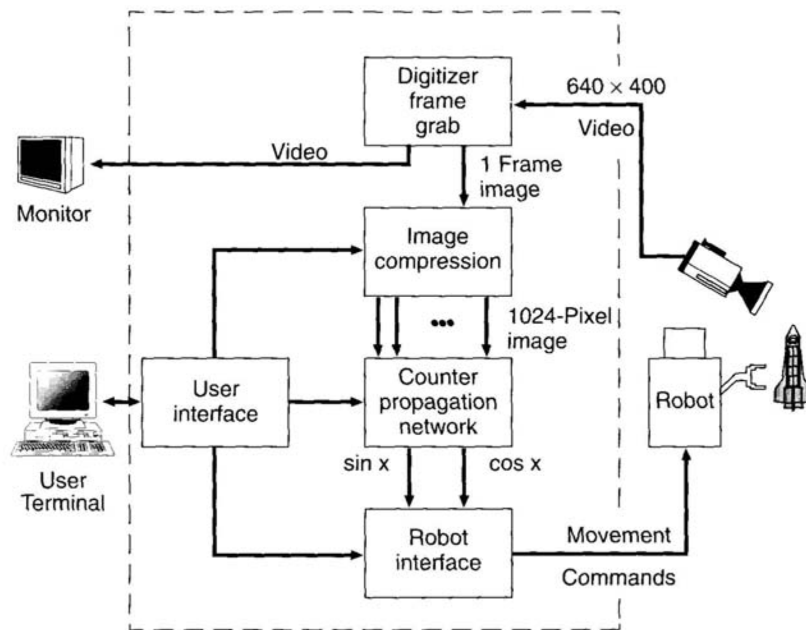


Figure 6.22 The system architecture for the spacecraft orientation system is shown. The video camera and frame-grabber capture a 256-by-256-pixel image of the model. That image is reduced to 32-by-32 pixels by a pixel-averaging technique, and is then thresholded to produce a binary image. The resulting 1024-component vector is used as the input to the neural network, which responds by giving the sine and cosine of the rotation angle of the principal axis of the model. These output values are converted to an angle that is sent as part of a command string to a mechanical robot assembly. The command sequence causes the robot to reach out and pick up the model. The angle is used to roll the robot's wrist to the proper orientation, so that the robot can grasp the model perpendicular to the long axis. *Source: Reprinted with permission from James A. Freeman, "Neural networks for machine vision: the spacecraft orientation demonstration." exponent: Ford Aerospace Technical Journal, Fall 1988.*

It would seem that this network is limited to classifying all input patterns into only one of 12 categories. An input pattern representing a rotation of 32 degrees, for example, probably would be classified as a 30-degree pattern by this network. One way to remedy this deficiency would be to add more units on the middle layer, allowing for a finer categorization of the input images. An alternative approach is to allow the output units to perform an interpolation for patterns that do not match one of the training patterns to within a certain tolerance. For this interpolative scheme to be accomplished, more than one unit on the competitive layer must share in winning for each input vector.

Recall that the output-layer units calculate their output values according to Eq. (6.23): $y'_k = \sum_j w_{kj} z_j$. In the normal case, where the ith hidden unit wins, $y'_k = w_{ki}$, since $z_j = 1$ for $j = i$ and $z_j = 0$ otherwise. Suppose two competitive units shared in winning—the ones with the two closest matching patterns. Further, let the output of those units be proportional to how close the input pattern is; that is, $z_j \propto \cos\theta_j$ for the two winning units. If we restrict the total outputfrom the middle layer to unity, then the output values from the output layer would be

$$y'_k = w_{ki} z_i + w_{kj} z_j$$

where the ith and jth units on the middle layer were the winners, and

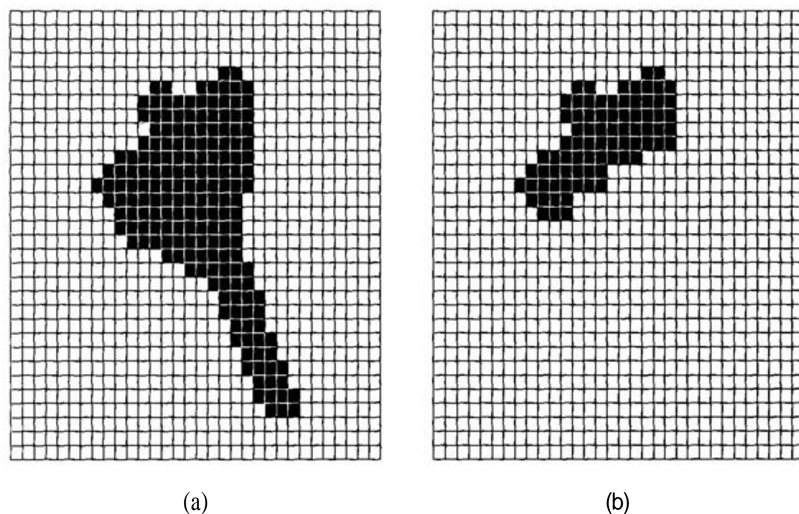
$$z_i + z_j = 1$$

The network output is a linear interpolation of the outputs that would be obtained from the two patterns that exactly matched the two hidden units that shared the victory.

Using this technique, the network will classify successfully input patterns representing rotation angles it had never seen during the training period. In our experiments, the average error was approximately ±3°. However, since a simple linear interpolation scheme is used, the error varied from almost 0 to as much as 10 degrees. Other interpolation schemes could result in considerably higher accuracy over the entire range of input patterns.

One of the benefits of using the neural-network approach to pattern matching is robustness in the presence of noise or of contradictory data. An example is shown in Figure 6.23, where the network was able to respond correctly, even though a substantial portion of the image was obscured.

It is unlikely that someone would use a neural network for a simple orientation determination. The methodology can be extended to more realistic cases, however, where the object can be rotated in three dimensions. In such cases, the time required to construct and train a neural network may be significantly less than the time required for development of algorithms that perform the identical tasks.



(a) (b)

Figure 6.23 These figures show 32-by-32-pixel arrays of two different input vectors for the spacecraft-orientation system. (a) This is a bit-mapped image of the space-shuttle model at an angle of 150° as measured clockwise from the vertical. (b) The obscured image was used as an input vector to the spacecraft-orientation system. The CPN responded with an angle of 149°. *Source: Reprinted with permission from James A. Freeman, "Neural networks for machine vision: the spacecraft orientation demonstration."* exponent: Ford Aerospace Technical Journal, *Fall 1988.*

6.4 THE CPN SIMULATOR

Even though it utilizes two different learning rules, the CPN is perhaps the least complex of the layered networks we will simulate, primarily because of the aspect of competition implemented on the single hidden layer. Furthermore, if we assume that the host computer system ensures that all input pattern vectors are normalized prior to presentation to the network, it is only the hidden layer that contains any special processing considerations: the input layer is simply a fan-out layer, and each unit on the output layer merely performs a linear summation of its active inputs. The only complication in the simulation is the determination of the *winning* unit(s), and the generation of the appropriate output for each of the hidden-layer units. In the remainder of this section, we will describe the algorithms necessary to construct the restricted CPN simulator. Then, we shall describe the extensions that must be made to implement the complete CPN. We conclude the chapter with thoughts on alternative methods of initializing and training the network.

6.4.1 The CPN Data Structures

Due to the similarity of the CPN simulator to the BPN discussed in Chapter 3, we will use those data structures as the basis for the CPN simulator. The only modification we will require is to the top-level network record specification. The reason for this modification should be obvious by now; since we have consistently used the network record as the repository for all network specific parameters, we must include the CPN-specific data in the CPN's top level declaration. Thus, the CPN can be defined by the following record structure:

```
record CPN =
   INPUTS : ^layer;   {pointer to input layer record}
   HIDDENS : ^layer;  {pointer to hidden layer record}
   OUTPUTS : ^layer;  {pointer to output layer record}
   ALPHA : float;     {Kohonen learning parameter}
   BETA : float;      {Grossberg learning parameters}
   N : integer;       {number of winning units allowed}
end record;
```

where the layer record and all lower-level structures are identical to those defined in Chapter 3. A diagram illustrating the complete structure defined for this network is shown in Figure 6.24.

6.4.2 CPN Algorithms

Since forward signal propagation through the CPN is easiest to describe, we shall begin with that aspect of our simulator. Throughout this discussion, we will assume that

- The network simulator has been initialized so that the internal data structures have been allocated and contain valid information
- The user has set the outputs of the network input units to a normalized vector to be propagated through the network
- Once the network generates its output, the user application reads the output vector from the appropriate array and uses that output accordingly

Recall from our discussion in Section 6.2.1 that processing in the CPN essentially starts in the hidden layer. Since we have assumed that the input vector is both normalized and available in the network data structures, signal propagation begins by having the computer calculate the total input stimulation received by each unit on the hidden layer. The unit (or units, in the case where $N > 1$) with the largest aggregate input is declared the winner, and the output from that unit is set to 1. The outputs from all losing units are simultaneously set to 0.

Once processing on the hidden layer is complete, the network output is calculated by performance of another sum-of-products at each unit on the output layer. In this case, the dot product between the connection weight vector to the unit in question and the output vector formed by all the hidden-layer units is

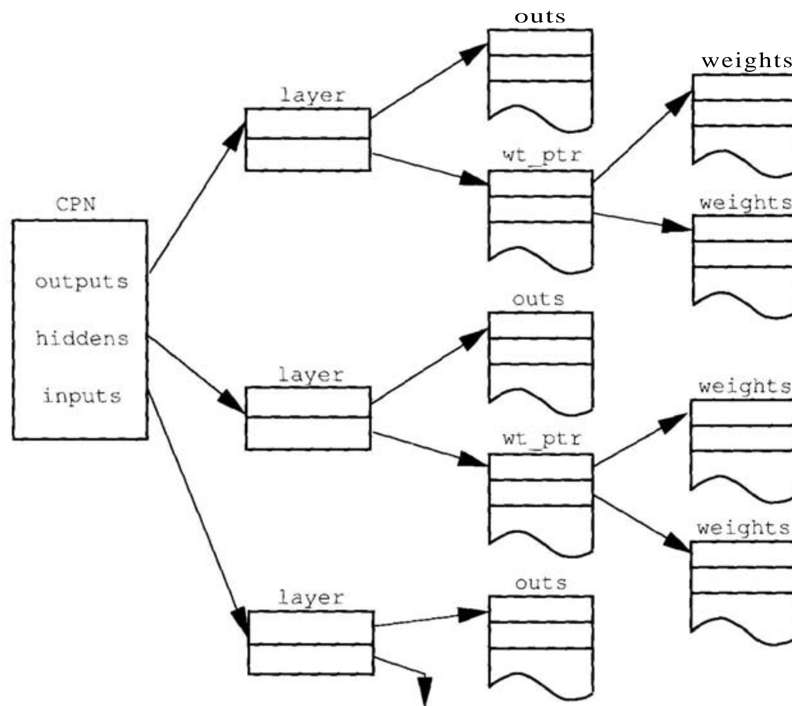


Figure 6.24 The complete data structure for the CPN is shown. These structures are representative of all the layered networks that we simulate in this text.

computed and used directly as the output for that unit. Since the hidden layer in the CPN is a competitive layer, the input computation at the output layer takes on a significance not usually found in an ANS; rather than combining feature indications from many units, which may be either excitatory or inhibitory (as in the BPN), the output units in the CPN are merely recalling features as *stored in the connections* between the winning hidden unit(s) and themselves. This aspect of memory recall is further illustrated in Figure 6.25.

Armed with this knowledge of network operation, there are a number of things we can do to make our simulation more efficient. For example, since we know that only a limited number of units (normally only one) in the hidden layer will be allowed to win the competition, there is really no point in forcing the computer to calculate the total input to every unit in the output layer. A much more efficient approach would be simply to allow the computer to *remember* which hidden layer unit(s) won the competition, and to restrict the

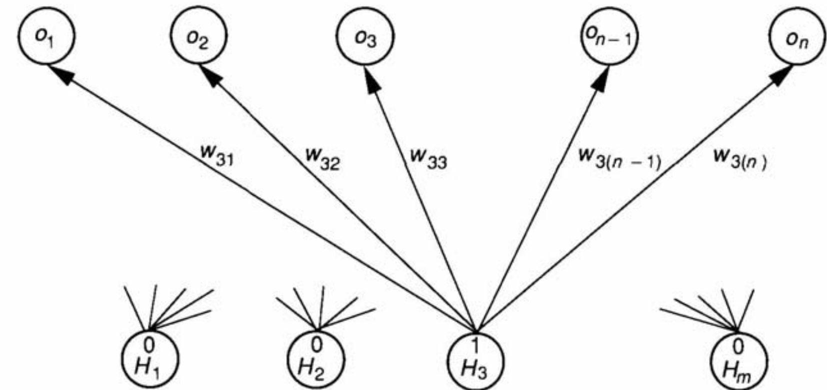


Figure 6.25 This figure shows the process of information recall in the output layer of the CPN. Each unit on the output layer receives an active input only from the winning unit(s) on the hidden layer. Since the connections between the winning hidden unit and each unit on the output layer contain the output value that was associated with the input pattern that won the competition during training, the process of computing the input at each unit on the output layer is nothing more than a selection of the appropriate output pattern from the set of available patterns stored in the input connections.

input calculation at each output unit to that unit's connections to the winning unit(s).

Also, we can consider the process of determining the winning hidden unit(s). In the case where only one unit is allowed to win ($N = 1$), determining the winner can be done easily as part of calculating the input to each hidden-layer unit; we simply need to compare the input just calculated to the value saved as the previously largest input. If the current input exceeds the older value, the current input replaces the older value, and processing continues with the next unit. After we have completed the input calculation for all hidden-layer units, the unit whose output matches the largest value saved can be declared the winner.[2]

On the other hand, if we allow more than one unit to win the competition ($N > 1$), the problem of determining the winning hidden units is more complicated. One problem we will encounter is the determination of *how many* units will be allowed to win simultaneously. Obviously, we will never have to allow all hidden units to win, but for how many possible winners must we account in

[2] This approach ignores the case where ties between hidden-layer units confuse the determination of the winner. In such an event, other criteria must be used to select the winner.

our simulator design? Also, we must address the issue of ranking the hidden-layer units so that we may determine which unit(s) had a greater response to the input; specifically, should we simply process all the hidden-layer units first, and sort them afterward, or should we attempt to rank the units as we process them?

The answer to these questions is truly application dependent; for our purposes, however, we will assume that we must account for no more than three winning units ($0 < N < 4$) in our simulator design. This being the case, we can also assume that it is more efficient to keep track of up to three winning units as we go, rather than trying to sort through all hidden units afterward.

CPN Production Algorithms. Using the assumptions described, we are now ready to construct the algorithms for performing the forward signal propagation in the CPN. Since the processing on each of the two active layers is different (recall that the input layer is fan-out only), we will develop two different signal-propagation algorithms: `prop_to_hidden` and `prop_to_output`.

```
procedure prop_to_hidden
 (NET:CPN; FIRST,SECOND,THIRD:INTEGER)
 {propagate to hidden layer, returning indices to
                                        3 winners}

var units : ^float[];          {pointer to unit outputs}
    invec : ^float[];          {pointer to input units}
    connects : ^float[];  {pointer to connection array}
    best : float;              {the current best match}
    i, j : integer;               {iteration counters}

begin
  best = -100;                 {initialize best choice}
  units = NET.HIDDENS^.OUTS;      {locate output array}

  for i = 1 to length (units)
    do                            {for all hidden units}
      units [i] = 0;           {initialize accumulator}
      invec = NET.INPUTS^.OUTS;     {locate input array}
      connects = NET.HIDDENS^.WEIGHTS[i]^;
                                         {locate inputs}

      for j = 1 to length (connects) do
          units [i] = units [i] + connects[j] * invec[j];
      end do;

      rank (units[i], FIRST, SECOND, THIRD);
    end do;

  compete (NET.HIDDENS^.OUTS, FIRST, SECOND, THIRD);
end procedure;
```


This procedure makes calls to two as-yet-undefined routines, `rank` and `compete`. The purpose of these routines is to sort the current input with the current best three choices, and to generate the appropriate output for all units in the specified layer, respectively. Because the design of the `rank` procedure is fairly straightforward, it is left to you as an exercise. On the other hand, the `compete` process must do the right thing no matter how many winners are allowed, making it somewhat involved. We therefore present the design for `compete` in its entirety.

```
procedure compete
 (UNITS:^float[]; FIRST,SECOND,THIRD:INTEGER)
 {generate outputs for all UNITS using competitive
                                         function}

var outputs : ^float[];    {step through output array}
    sum : float;                 {local accumulator}
    win, place, show : float;        {store outputs}
    i : integer;                {iteration counter}

begin
  outputs = UNITS;               {locate output array}
  sum = outputs[FIRST];      {initialize accumulator}
  win = outputs[FIRST];         {save winning value}

  if (SECOND != 0)              {if a second winner}
  then                         {add its contribution}
    sum = sum + outputs[SECOND];
    place = outputs[SECOND];  {save second place value}

    if (THIRD != 0)          {if a third place winner}
    then                       {add its contribution}
      sum = sum + outputs[THIRD];
      show = outputs[THIRD];   {save third place value}
    end if;
  end if;

  for i = 1 to length (units)     {for all hidden units}
    do
      outputs[i] = 0;             {set outputs to zero}
    end do;

  outputs[FIRST] = win / sum;      {set winners output}

  if (SECOND != 0)           {now update second winner}
  then
    outputs[SECOND] = place / sum;

    if (THIRD != 0)                   {and third place}
    then
      outputs[THIRD] = show / sum;
    end if;
  end if;
end procedure;
```

Before we move on to the prop_to_output routine, you should note that the compete procedure relies on the fact that the values of SECOND and THIRD are nonzero if and only if more than one unit wins the competition. Since it is assumed that these values are set as part of the rank procedure, you should take care to ensure that these variables are manipulated according to the number of winning units indicated by the value in the NET.N variable.

Let us now consider the process of propagating information to the output layer in the CPN. Once we have completed the signal propagation to the hidden layer, the outputs on the hidden layer will be nonzero only from the winning units. As we have discussed before, we could now proceed to perform a complete input summation at every unit on the output layer, but that would prove to be needlessly time-consuming. Since we have designed the prop_to_hidden procedure to return the index of the winning unit(s), we can assume that the top-level routine to propagate information through the network completely will have access to that information prior to calling the procedure to propagate information to the output layer. We can therefore code the prop_to_output procedure so that only those connections between the winning units and the output units are processed. Also, notice that the successful use of this procedure relies on the values of the SECOND and THIRD variables being nonzero only if more than one winner was allowed.

```
procedure prop_to_output
 (NET:CPN; FIRST,SECOND,THIRD:INTEGER)
 {generate outputs for units on the output layer}

var units : ^float[];              {locate output units}
    hidvec : ^float[];             {locate hidden units}
    connects : ^float[];            {locate connections}
    i : integer;                     {iteration counter}

begin
  units = NET.OUTPUTS^.OUTS;    {start of output array}
  hidvec = NET.HIDDENS^.OUTS;   {start of hidden array}

  for i = 1 to length (units)    {for all output units}
   do
    connects = NET.OUTPUTS^.WEIGHTS[i]^;
    units[i] = hidvec[FIRST] * connects[FIRST];

    if (SECOND != 0)
                  {if there is a second winning unit}
    then
      units[i] = units[i] + hidvec[SECOND]
                          * connects[SECOND];

      if (THIRD != 0)
                   {if there is a third winning unit}
```



```
      then
        units[i] = units[i] + hidvec[THIRD]
                            * connects[THIRD];
      end if;
    end if;
  end do;
end procedure;
```

You may now be asking yourself how we can be assured that the hidden-layer units will be using the appropriate output values for any number of winning units. To answer this question, we recall that we specified that at least one unit is guaranteed to win, and that at most, three will share in the victory. Inspection of the compete and prop_to_output routines shows that, with only one winning unit, the output of all non-winning units will be 0, whereas the winning unit will generate a 1. As we increase the number of units that we allow to win, the strength of the output from each of the winning units is proportionally decreased, so that the relative contribution from all winning units will linearly interpolate between output patterns the network was trained to produce.

Now we are prepared to define the top-level algorithm for forward signal propagation in the CPN. As before, we assume that the input vector has been set previously by an application-specific input routine.

```
procedure propagate (NET:CPN)
 {perform a forward signal propagation in the CPN}

var first,
    second
    third : integer;         {indices for winning units}

begin
  prop_to_hidden (NET, first, second, third);
  prop_to_output (NET, first, second, third);
end procedure;
```

CPN Learning Algorithms. There are two significant differences between forward signal propagation and learning in the CPN: during learning, only one unit on the hidden layer can win the competition, and, quite obviously, the network connection weights are updated. Yet, even though they are different, much of the activity that must be performed during learning is identical to the forward signal propagation. As you will see, we will be able to reuse the production-mode algorithms to a large extent as we develop our learning-mode procedures.

We shall begin by training the hidden-layer units to recognize our input patterns. Having completed that activity, we will proceed to train the output layer to reproduce the target outputs from the specified inputs. Let us first consider the process of training the hidden layer in the CPN. Assuming the input layer units have been initialized to contain a normalized vector to be

learned, we can define the learning algorithm for the hidden-layer units in the following manner.

```
procedure learn_input (NET:CPN)
{update the connections to the winning hidden unit}

var winner : integer;      {used to locate winning unit}
    dummy1, dummy2 : integer;
                  {dummy variables for second and third}
    connects : ^float[];     {locate connection array}
    units : ^float[];              {locate input array}
    i : integer;                   {iteration counter}

begin
  dummy1 = 0;                    {no need for second or}
  dummy2 = 0;                       {third winning unit}
  prop_to_hidden (NET, winner, dummy1, dummy2);
  units = NET.INPUTS^.OUTS;        {locate input array}
  connects = NET.HIDDENS^.WEIGHTS[winner]^;
                                   {locate connections}

  for i = 1 to length (connects)
                       {for all connections to winner}
    do
      connects[i] = connects[i] +
                    NET.ALPHA * (units[i] - connects[i]);
    end do;
end procedure;
```

Notice that this algorithm has no access to information that would indicate when the competitive layer has been trained sufficiently. Unlike in many of the other networks we have studied, there is no error measure to indicate convergence. For that reason, we have chosen to design this training algorithm so that it performs only one training pass in the competitive layer. A higher-level routine to train the entire network must therefore be coded such that it can reasonably determine when the competitive layer has completed its training.

Moving on to the output layer, we will construct our training algorithm for this part of the network such that only those connections from the winning hidden unit to the output layer are adapted. This approach will allow us to complete the training of the competitive layer before starting the training on the accretive layer. Hopefully, the use of this approach will enable the CPN to classify inputs correctly as it is training outputs, to avoid confusing the network.

```
procedure learn_output (NET:CPN; TARGET:^float[])
{train the output layer to reproduce the specified
                                               vector}

var winner : integer;      {used to locate winning unit}
    dummy1, dummy2 : integer;
                  {dummy variables for second & third}
```



```
    connects : ^float[];     {locate connection array}
    units : ^float[];             {locate output array}
    i : integer;                    {iteration counter}

begin
  dummy1 = 0;                     {no need for second or}
  dummy2 = 0;                        {third winning unit}

  prop_to_hidden (NET, winner, dummy1, dummy2);
  units = NET.OUTPUTS^.OUTS;      {locate output array}

  for i = 1 to length (units)   {for all output units}
    do
      connects = NET.OUTPUTS^.WEIGHTS^[i];
                                   {locate connections}
      connects[winner] = connects[winner] +
                         NET.BETA * (TARGET[i] -
                                     connects[winner]);
    end do;
end procedure;
```

As with learn_input, learn_output performs only one training pass and makes no assessment of error. When the CPN is used, it is the application that makes the determination as to when training is complete. As an example, consider the spacecraft-orientation system described in Section 6.3. This network was constructed to learn the image of the space shuttle at 12 different orientations, producing the scaled sine and cosine of the angle between the reference position and the image position as output. Using our CPN algorithms, the training process for this application might have taken the following form:

```
procedure learn (NET:CPN; IMAGEFILE:disk file)
{teach the NET using data in the IMAGEFILE}

var iopairs : array [1..12,1..1026] of float;
    target : array [1..2] of float;
    status : array [1..12] of boolean;
    done : boolean;
    i, j : integer;

begin
  NET.N = 1;                       {force only one winner}
  READ_INPUT_FILE (IMAGEFILE, iopairs[1,1]);
                                             {init array}

  done = false;                     {train at least once}
  SET_FALSE (status);           {initialize status array}

  while (not done)              {until training complete}
    do
      for i = 1 to 12            {for each training pair}
```

```
          do
            j = random (12);
                            {select a pattern at random}
            SET_INPUT (CPN, iopairs[j, 1] ;
            learn_input (CPN); {train competitive layer}
          end do;

        TEST_IF_INPUT_LEARNED (status, done);
      end do;

    done = false;           {train output at least once}
    SET_FALSE (status);        {initialize status array}

    while (not done)           {until training complete}
      do
        for i = 1 to 12          {for each training pair}
          do                      {train accretive layer}
            SET_INPUT (CPN, iopairs[i, 1];
            GET_TARGET (target, iopairs[i, 1025]);
            learn_output (CPN, target);
          end do;

        TEST_IF_OUTPUT_LEARNED (Status, done);
      end do;
  end procedure;
```

where the application provided routines `test_if_input_learned` and `test_if_output_learned` perform the function of deciding when the CPN has been trained sufficiently. In the case of testing the competitive layer, this determination was accomplished by verifying that all 12 input patterns caused different hidden-layer units to win the competition. Similarly, the output test indicated success when no output unit generated an actual output different by more than 0.001 from the desired output for each of the 12 patterns. The other routines used in the application, `READ_INPUT_FILE`, `SET_FALSE`, `SET_INPUT`, and `GET_TARGET`, merely perform housekeeping functions for the system. The point is, however, that there is no general heuristic that we can use to determine when to stop training the CPN simulator. If we had had many more training pairs than hidden-layer units, the functions performed by `TEST_IF_INPUT_LEARNED` and `TEST_IF_OUTPUT_LEARNED` might have been altogether different.

6.4.3 Simulating the Complete CPN

Now that we have completed our discussion of the restricted CPN, let us turn our attention for a moment to the full CPN. In terms of simulating the complete network, there are only two differences between the restricted and complete network implementations:

1. The size of the network, in terms of number of units and connections
2. The use of the network from the applications perspective

Quite obviously, the number of units in the network has grown from $N + H + M$, where N and M specify the number of units in the input and output layers, respectively, to $2(N+M) + H$. Similarly, the number of connections that must be maintained has doubled, expanding from $H(N + M)$ to $2H(N + M)$. Therefore, the extension from the restricted to the complete CPN has slightly less than doubled the amount of computer memory needed to simulate the network. In addition, the extra units and connections place an enormous overhead on the amount of computer time needed to perform the simulation, in that there are now $N + M$ extra connections to be processed at every hidden unit.

As illustrated in Figure 6.26, the complete CPN requires no modification to the algorithms we have just developed, other than to present both the input and output patterns as target vectors for the output layer. This assertion holds true assuming the user abides by the observation that, when going from input to output, the extra M units on the input layer are *zeroed* prior to performing the signal propagation. This being the case, the inputs from the extra units contribute nothing to the dot-product calculation at each hidden unit, effectively eliminating them from consideration in determining the winning unit. By the same token, the original N units must be zeroed prior to performance of the Counterpropagation from the M new units to recover the original input.

6.4.4 Practical Considerations for the CPN Simulator

We earlier promised a discussion of the practical considerations of which we might take advantage when simulating the CPN. We shall now live up to that promise by offering insights into improving the performance of the CPN simulator.

Many times, a CPN application will require the network to function as an associative memory; that is, we expect the network to recall a specific output when presented with an input that is similar to a training input. Such an input could be the original input with noise added, or with part of the input missing. When constructing a CPN to act in this capacity, we usually create it with *as many hidden units as there are items to store.* In doing so, and in allowing only one unit to win the competition, we ensure that the network will always generate the exact output that was associated with the training input that most closely resembles the current input.

Having made this observation, we can now see how it is possible to reduce the amount of time needed to train the CPN by eliminating the need to train the competitive layer. We can do this reduction by *initializing* the connections to each hidden unit such that each input training pattern is mapped onto the connections of only one hidden unit. In essence, we will have trained the competitive layer by initializing the connections, in a manner similar to the process of initializing the BAM, described in Chapter 4. All that remains from

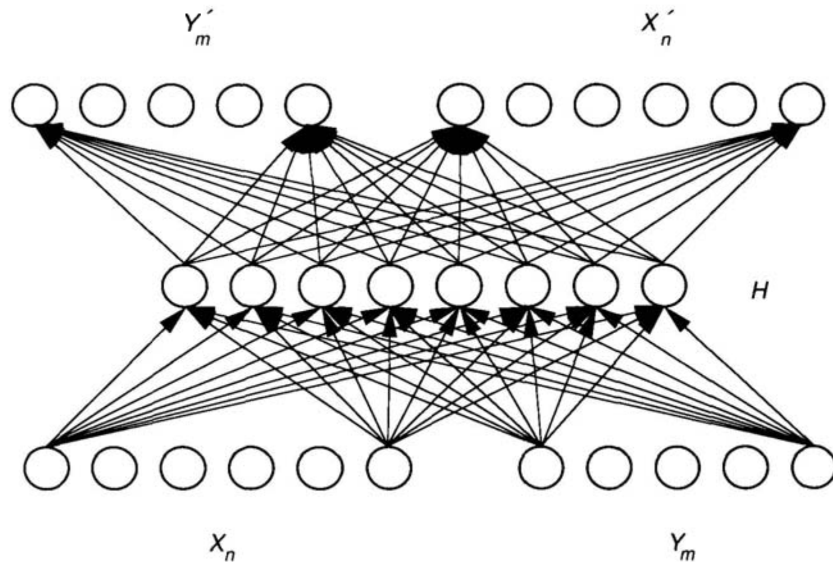


Figure 6.26 The complete CPN processing model is shown. Using the algorithms developed for the limited CPN, processing begins with the application of an input pattern on the X_n units *and* by zeroing of the extra Y_m units, which we use to represent the output. In this case, the output units (Y'_m) will produce the output pattern associated with the given input pattern during training. Now, to produce the counterpropagation effect, the opposite situation occurs. The given output pattern is applied to the input units previously zeroed (Y_m), while the X_n units are zeroed. The output of the network, on the X'_n units, represent the input pattern associated with the given output pattern during training.

this point is the training of the output layer, which ought to conclude in fairly short order. An example of this type of training is shown in Figure 6.27.

Another observation about the operation of the CPN can provide us with insight into improving the ability of the network to discriminate between similar vectors. As we have discussed, the competitive layer acts to select between one of the many input patterns the network was trained to recognize. It does this selection by computing the dot product between the input vector, I, and the connection weight vector, **w**. Since these two vectors are normalized, the resulting value represents the cosine of the angle between them in n-space. However, this approach can lead to problems if we allow the use of the null vector, 0, as a valid input. Since 0 cannot be normalized, another method of

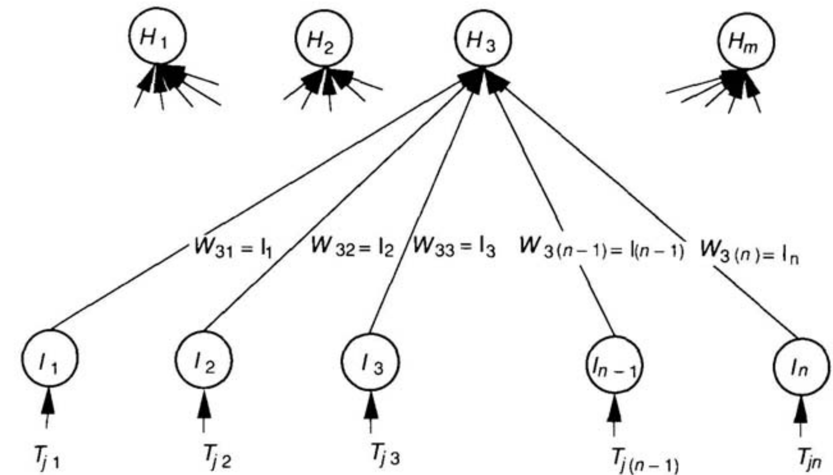


Figure 6.27 A restricted CPN initialized to eliminate the training of the competitive layer is shown. Note that the number of hidden units is exactly equal to the number of training patterns to be learned.

propagating signals to the hidden layer must be found. One alternative method that has shown promising results is to use the magnitude of the difference vector between the unnormalized I and **w** vectors as the input computation for each hidden unit. Specifically, let

$$\text{net}_i = (\sum_{j=0}^{n}(|W_{ij} - I_j|)^2)^{\frac{1}{2}}$$

be the input activation calculation performed at each hidden unit, rather than the traditional sum-of-products.

Use of this method prevents the CPN from creating duplicate internal mappings for similar, but different, input vectors. It also allows use of the null vector as a valid training input, something that we have found to be quite useful. Finally, at least one other researcher has indicated that this alternative method can improve the ability of the network to learn by reducing the number of training presentations needed to have the network classify input patterns properly [11].

Programming Exercises

6.1. Implement the CPN simulator described in the text and test it by training it to recognize the 36 uppercase ASCII characters from a pixel matrix representing the image of the character. For example, the 6×5 matrix illustrated next represents the pixel image of the character A. The equivalent ASCII

code for that character is 41_{16}. Thus, the ordered pair $(08A8FE31_{16}, 41_{16})$ represent one training pair for the network. Complete this example by training the network to recognize the other 35 alphanumeric characters from their pixel image. How many hidden-layer units are needed to precisely recall all 36 characters'?

```
..x..
.x.x.
x...x
xxxxx   =   00100 01010 10001 11111 10001 10001
x...x
x...x
```

6.2. Without resizing the network, retrain the CPN described in Programming Exercise 6.1 to recognize both the upper- and lowercase ASCII alphabetic characters. Describe how accurately the CPN identifies all characters after training. Include in your discussion any reasons you might have to explain why the CPN misclassifies some characters.

6.3. Repeat Programming Exercise 6.1, but allow two hidden units to win the competition during production. Explain the network's behavior under these circumstances.

6.4. Recreate the spacecraft-orientation example in Section 6.3 using your CPN simulator. You may simplify the problem by using a smaller matrix to represent the video image. For example, the 5×5 matrix shown next might be used to represent the shuttle image in the vertical position. Train the network to recognize your image at 45-degree rotational increments around the circle. Let two units win the competition, as in the example, and let the two output units produce the scaled sine and cosine of the image angle. Test your network by obscuring the image (enter a vector with Os in place of 1s), and describe the results.

```
..x..
..x..
..x..
.xxx.   =   00100 00100 00100 01110 11111
xxxxx
```

6.5. Describe what would happen in Programming Exercise 6.4 if you only allowed one unit to win the competition. Describe what would happen if three units were allowed to win.

6.6. Implement the complete CPN simulator using the guidelines provided in the text. Train the network using the spacecraft-orientation data, and exercise the simulator in both the forward-propagation and counterpropagation modes. How well does the simulator produce the desired *input* pattern when given sine and cosine values that are not on a 45 degree angle?

Suggested Readings

The papers and book by Hecht-Nielsen are good companions to the material in this chapter on the CPN [5, 6, 8]. Hecht-Nielsen also has a paper that discusses some applications areas appropriate to the CPN [7].

The instar, outstar, and avalanche networks are discussed in detail in the papers by Grossberg in the collection *Studies of Mind and Brain* [4]. Individual papers from this collection are listed in the bibliography.

Bibliography

[1] James A. Freeman. Neural networks for machine vision applications: The spacecraft orientation demonstration. *e^x ponent: Ford Aerospace Technical Journal*, pp. 16–20, Fall 1988.

[2] Stephen Grossberg. How does a brain build a cognitive code. In Stephen Grossberg, editor, *Studies of Mind and Brain.* D. Reidel Publishing, Boston, pp. 1-52, 1982.

[3] Stephen Grossberg. Learning by neural networks. In Stephen Grossberg, editor, *Studies of Mind and Brain.* D. Reidel Publishing, Boston, pp. 65-156, 1982.

[4] Stephen Grossberg, editor. *Studies of Mind and Brain,* volume 70 *of Boston Studies in the Philosophy of Science.* D. Reidel Publishing Company, Boston, 1982.

[5] Robert Hecht-Nielsen. Counterpropagation networks. *Applied Optics,* 26(23):4979–4984, December 1987.

[6] Robert Hecht-Nielsen. Counterpropagation networks. In Maureen Caudill and Charles Butler, editors, *Proceedings of the IEEE First International Conference on Neural Networks,* Piscataway, NJ, pages II-19-II-32, June 1987. IEEE.

[7] Robert Hecht-Nielsen. Applications of Counterpropagation networks. *Neural Networks,* 1(2):131-139, 1988.

[8] Robert Hecht-Nielsen. *Neurocomputing.* Addison-Wesley, Reading, MA, 1990.

[9] David Hestenes. How the brain works: The next great scientific revolution. Presented at the Third Workshop on Maximum Entropy and Bayesian Methods in Applied Statistics, University of Wyoming, August 1983.

[10] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In David E. Rumelhart and James L. McClelland, editors, *Parallel Distributed Processing,* Chapter 8. MIT Press, Cambridge, MA, pp. 318-362, 1986.

[11] Donald Woods. Back and counter propagation aberrations. In *Proceedings of the IEEE First International Conference on Neural Networks,* pp. I-473–I-479, IEEE, San Diego, CA, June 1987.