11 MACHINE LEARNING: CONNECTIONIST

A cat that once sat on a hot stove will never again sit on a hot stove or on a cold one either. . .

—MARK TWAIN

Everything is vague to a degree you do not realize till you have tried to make it precise. . .

—BERTRAND RUSSELL

. . . as if a magic lantern threw the nerves in patterns on a screen. . .

—T. S. ELIOT, *The Love Song of J. Alfred Prufrock*

11.0 Introduction

In Chapter 10 we emphasized a symbol-based approach to learning. A central aspect of this hypothesis is the use of symbols to refer to objects and relations in a domain. In the present chapter, we introduce *neurally* or *biology* inspired approaches to learning.

Neurally inspired models, sometimes known as *parallel distributed processing (PDP)* or *connectionist* systems, de-emphasize the explicit use of symbols in problem solving. Instead, they hold that intelligence arises in systems of simple, interacting components (biological or artificial neurons) through a process of learning or adaptation by which the connections between components are adjusted. Processing in these systems is distributed across collections or layers of neurons. Problem solving is parallel in the sense that all the neurons within the collection or layer process their inputs simultaneously and independently. These systems also tend to degrade gracefully because information and processing are distributed across the network's nodes and layers.

In connectionist models there is, however, a strong representational character both in the creation of input parameters as well as in the interpretation of output values. To build a

neural network, for example, the designer must create a scheme for encoding patterns in the world into numerical quantities in the net. The choice of an encoding scheme can play a crucial role in the eventual success or failure of the network to learn.

In connectionist systems, processing is parallel and distributed with no manipulation of symbols as symbols. Patterns in a domain are encoded as numerical vectors. The connections *between* components, are also represented by numerical values. Finally, the transformation of patterns is the result of a numerical operations, usually, matrix multiplications. These "designer's choices" for a connectionist architecture constitute the *inductive bias* of the system.

The algorithms and architectures that implement these techniques are usually trained or conditioned rather than explicitly programmed. Indeed, this is a major strength of the approach: an appropriately designed network architecture and learning algorithm can often capture invariances in the world, even in the form of strange attractors, without being explicitly programmed to recognize them. How this happens makes up the material of Chapter 11.

The tasks for which the neural/connectionist approach is well suited include:

classification, deciding the category or grouping to which an input value belongs;
pattern recognition, identifying structure or pattern in data;
memory recall, including the problem of content addressable memory;
prediction, such as identifying disease from symptoms, causes from effects;
optimization, finding the "best" organization of constraints; and
noise filtering, or separating signal from background, factoring out the irrelevant components of a signal

The methods of this chapter work best on those tasks that can be difficult to formulate for symbolic models. This typically includes tasks in which the problem domain requires perception-based skills, or lacks a clearly defined syntax.

In Section 11.1 we introduce neurally inspired learning models from an historical viewpoint. We present the components of neural network learning, including the "mechanical" neuron, and describe some historically important early work, including the McCulloch–Pitts (1943) neuron. The evolution of the network training paradigms over the past 60 years offers important insights into the present state of the discipline.

In Section 11.2, we continue the historical presentation with the introduction of *perceptron* learning, and the *delta* rule. We present an example of the perceptron used as a classifier. In Section 11.3 we introduce neural nets with hidden layers, and the *backpropagation* learning rule. These innovations were introduced for artificial neural networks to overcome problems the early systems had in generalizing across data points that were not linearly separable. Backpropagation is an algorithm for apportioning "blame" for incorrect responses to the nodes of a multilayered system with continuous thresholding.

In Section 11.4 we present models for *competitive learning* developed by Kohonen (1984) and Hecht-Nielsen (1987). In these models, network weight vectors are used to represent patterns rather than connection strengths. The *winner-take-all* learning algorithm selects the node whose pattern of weights is most like the input vector and adjusts it to make it more like the input vector. It is unsupervised in that *winning* is simply

identifying the node whose current weight vector most closely resembles the input vector. The combination of Kohonen with Grossberg (1982) layers in a single network offers an interesting model for stimulus–response learning called *counter propagation* learning.

In Section 11.5 we present Hebb's (1949) model of reinforcement learning. Hebb conjectured that each time one neuron contributes to the firing of another neuron, the strength of the pathway between the neurons is increased. Hebbian learning is modeled by a simple algorithm for adjusting connection weights. We present both unsupervised and supervised versions of Hebbian learning. We also introduce the linear associator, a Hebbian based model for pattern retrieval from memory.

Section 11.6 introduces a very important family of networks called *attractor networks*. These networks employ feedback connections to repeatedly cycle a signal within the network. The network output is considered to be the network state upon reaching equilibrium. Network weights are constructed so that a set of *attractors* is created. Input patterns within an attractor *basin* reach equilibrium at that attractor. The attractors can therefore be used to store patterns in a memory. Given an input pattern, we retrieve either the closest stored pattern in the network or a pattern associated with the closest stored pattern. The first type of memory is called *autoassociative*, the second type *heteroassociative*. John Hopfield (1982), a theoretical physicist, defined a class of attractor networks whose convergence can be represented by energy minimization. Hopfield networks can be used to solve constraint satisfaction problems, such as the traveling salesperson problem, by mapping the optimization function into an energy function (Section 11.6.4).

In Chapter 12, we present evolutionary models of learning, including genetic algorithms and artificial life. Chapter 13 presents dynamic and stochastic models of learning. We discuss representational issues and bias in learning as well as the strengths of each learning paradigm in Section 16.3.

11.1 Foundations for Connectionist Networks

11.1.1 Early History

Connectionist architectures are often thought of as a recent development, however we can trace their origins to early work in computer science, psychology, and philosophy. John von Neumann, for example, was fascinated by both cellular automata and neurally inspired approaches to computation. Early work in neural learning was influenced by psychological theories of animal learning, especially that of Hebb (1949). In this section, we outline the basic components of neural network learning, and present historically important early work in the field.

The basis of neural networks is the artificial neuron, as in Figure 11.1. An artificial neuron consists of:

Input signals, x_i. These data may come from the environment, or the activation of other neurons. Different models vary in the allowable range of the input values; typically inputs are discrete, from the set $\{0, 1\}$ or $\{-1, 1\}$, or real numbers.

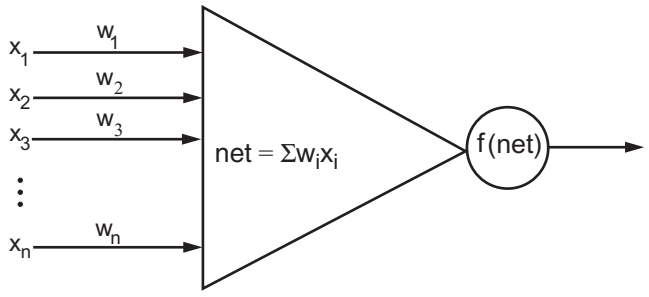


Figure 11.1 An artificial neuron, input vector x_i, weights on each input line, and a thresholding function f that determines the neuron s output value. Compare this figure with the actual neuron of Figure 1.2.

A set of real valued weights, w_i. The weights describe connection strengths.

An activation level $\Sigma w_i x_i$. The neuron's activation level is determined by the cumulative strength of its input signals where each input signal is scaled by the connection weight w_i along that input line. The activation level is thus computed by taking the sum of the weighted inputs, that is, $\Sigma w_i x_i$.

A threshold function, f. This function computes the neuron's final or output state by determining how far the neuron's activation level is below or above some threshold value. The threshold function is intended to produce the on/off state of actual neurons.

In addition to these properties of individual neurons, a neural network is also characterized by global properties such as:

The network topology. The topology of the network is the pattern of connections between the individual neurons. This topology is a primary source of the nets inductive bias.

The learning algorithm used. A number of algorithms for learning are presented in this chapter.

The encoding scheme. This includes the interpretation placed on the data to the network and the results of its processing.

The earliest example of neural computing is the McCulloch–Pitts neuron (McCulloch and Pitts 1943). The inputs to a McCulloch–Pitts neuron are either excitatory (+1) or inhibitory (−1). The activation function multiplies each input by its corresponding weight and sums the results; if the sum is greater than or equal to zero, the neuron returns 1, otherwise, −1. McCulloch and Pitts showed how these neurons could be constructed to compute any logical function, demonstrating that systems of these neurons provide a complete computational model.

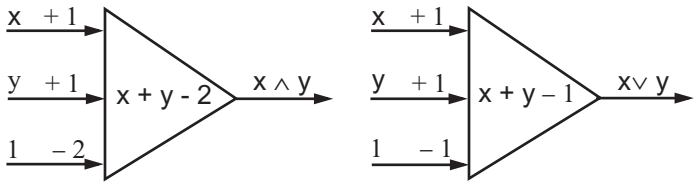


Figure 11.2 McCulloch–Pitts neurons to calculate the logic functions and and or.

Figure 11.2 shows the McCulloch-Pitts neurons for computing logical functions. The and neuron has three inputs: x and y are the values to be conjoined; the third, sometimes called a *bias*, has a constant value of +1. The input data and bias have weights of +1, +1, and –2, respectively. Thus, for any values of x and y, the neuron computes the value of x + y – 2: if this value is less than 0, it returns –1, otherwise a 1. Table 11.1 illustrates the neuron computing x ∧ y. In a similar fashion, the weighted sum of input data for the *or* neuron, see Figure 11.2, is greater than or equal to 0 unless both x and y equal –1.

x	y	x + y – 2	Output
1	1	0	1
1	0	–1	–1
0	1	–1	–1
0	0	–2	–1

Table 11.1 The McCulloch–Pitts model for logical and.

Although McCulloch and Pitts demonstrated the power of neural computation, interest in the approach only began to flourish with the development of practical learning algorithms. Early learning models drew heavily on the work of the psychologist D. O. Hebb (1949), who speculated that learning occurred in brains through the modification of synapses. Hebb theorized that repeated firings across a synapse increased its sensitivity and the future likelihood of its firing. If a particular stimulus repeatedly caused activity in a group of cells, those cells come to be strongly associated. In the future, similar stimuli would tend to excite the same neural pathways, resulting in the recognition of the stimuli. (See Hebb's actual description, Section 11.5.1.) Hebb's model of learning worked purely on reinforcement of used paths and ignored inhibition, punishment for error, or attrition. Modern psychologists attempted to implement Hebb's model but failed to produce general results without addition of an inhibitory mechanism (Rochester et al. 1988, Quinlan 1991). We consider the Hebbian model of learning in Section 11.5.

In the next section we extend the McCulloch–Pitts neural model by adding layers of connected neural mechanisms and algorithms for their interactions. The first version of this was called the *perceptron*.

11.2 Perceptron Learning

11.2.1 The Perceptron Training Algorithm

Frank Rosenblatt (1958, 1962) devised a learning algorithm for a type of single layer network called a *perceptron*. In its signal propagation, the perceptron was similar to the McCulloch–Pitts neuron, which we will see in Section 11.2.2. The input values and activation levels of the perceptron are either −1 or 1; weights are real valued. The activation level of the perceptron is given by summing the weighted input values, $\Sigma x_i w_i$. Perceptrons use a simple hard-limiting threshold function, where an activation above a threshold results in an output value of 1, and −1 otherwise. Given input values x_i, weights w_i, and a threshold, t, the perceptron computes its output value as:

$$\begin{array}{rl} 1 & \text{if } \Sigma x_i w_i \geq t \\ -1 & \text{if } \Sigma x_i w_i < t \end{array}$$

The perceptron uses a simple form of supervised learning. After attempting to solve a problem instance, a teacher gives it the correct result. The perceptron then changes its weights in order to reduce the error. The following rule is used. Let c be a constant whose size determines the learning rate and d be the desired output value. The adjustment for the weight on the ith component of the input vector, Δw_i, is given by:

$$\Delta w_i = c(d - \text{sign}(\Sigma x_i w_i))\, x_i$$

The $\text{sign}(\Sigma x_i w_i)$ is the perceptron output value. It is +1 or −1. The difference between the desired output and the actual output values will thus be 0, 2, or −2. Therefore for each component of the input vector:

If the desired output and actual output values are equal, do nothing.

If the actual output value is −1 and should be 1, increment the weights on the ith line by $2cx_i$.

If the actual output value is 1 and should be −1, decrement the weights on the ith line by $2cx_i$.

This procedure has the effect of producing a set of weights which are intended to minimize the average error over the entire training set. If there exists a set of weights which give the correct output for every member of the training set, the perceptron learning procedure will learn it (Minsky and Papert 1969).

Perceptrons were initially greeted with enthusiasm. However, Nils Nilsson (1965) and others analyzed the limitations of the perceptron model. They demonstrated that perceptrons could not solve a certain difficult class of problems, namely problems in which the data points are not linearly separable. Although various enhancements of the

perceptron model, including multilayered perceptrons, were envisioned at the time, Marvin Minsky and Seymour Papert, in their book *Perceptrons* (1969), argued that the linear separability problem could not be overcome by any form of the perceptron network.

An example of a nonlinearably separable classification is *exclusive-or*. Exclusive-or may be represented by the truth table:

x_1	x_2	Output
1	1	0
1	0	1
0	1	1
0	0	0

Table 11.2 The truth table for exclusive-or.

Consider a perceptron with two inputs, x_1, x_2, two weights, w_1, w_2, and threshold t. In order to learn this function, a network must find a weight assignment that satisfies the following inequalities, seen graphically in Figure 11.3:

$w_1*1 + w_2*1 < t$, from line 1 of the truth table.
$w_1*1 + 0 > t$, from line 2 of the truth table.
$0 + w_2*1 > t$, from line 3 of the truth table.
$0 + 0 < t$, or t must be positive, from the last line of the table.

This series of equations on w_1, w_2, and t has no solution, proving that a perceptron that solves *exclusive-or* is impossible. Although multilayer networks would eventually be built that could solve the exclusive-or problem, see Section 11.3.3, the perceptron learning algorithm only worked for single layer networks.

What makes exclusive-or impossible for the perceptron is that the two classes to be distinguished are not *linearly separable*. This can be seen in Figure 11.3. It is impossible to draw a straight line in two dimensions that separates the data points {(0,0), (1,1)} from {(0,1), (1,0)}.

We may think of the set of data values for a network as defining a space. Each parameter of the input data corresponds to one dimension, with each input value defining a point in the space. In the exclusive-or example, the four input values, indexed by the x_1, x_2 coordinates, make up the data points of Figure 11.3. The problem of learning a binary classification of the training instances reduces to that of separating these points into two groups. For a space of n dimensions, a classification is linearly separable if its classes can be separated by an $n - 1$ dimensional hyperplane. (In two dimensions an n-dimensional hyperplane is a line; in three dimension it is a plane, etc.).

As a result of the linear separability limitation, research shifted toward work in symbol-based architectures, slowing progress in the connectionist methodology. Subsequent work in the 1980s and 1990s has shown these problems to be solvable, however see (Ackley et al. 1985, Hinton and Sejnowski 1986, 1987).

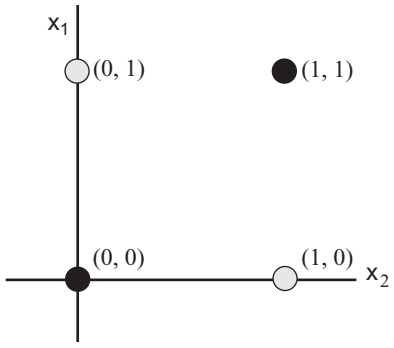


Figure 11.3 The exclusive-or problem. No straight line in two-dimensions can separate the (0, 1) and (1, 0) data points from (0, 0) and (1, 1).

In Section 11.3 we discuss *backpropagation*, an extension of perceptron learning that works for multilayered networks. Before examining backpropagation, we offer an example of a perceptron network that performs classifications. We end Section 11.2 by defining the *generalized delta rule*, a generalization of the perceptron learning algorithm that is used in many neural network architectures, including backpropagation.

11.2.2 An Example: Using a Perceptron Network to Classify

Figure 11.4 offers an overview of the classification problem. Raw data from a space of possible points are selected and transduced to a new data/pattern space. In this new pattern space features are identified, and finally, the entity these features represent is classified. An example would be sound waves recorded on a digital recording device. From there the acoustic signals are translated to a set of amplitude and frequency parameters. Finally, a classifier system might recognize these feature patterns as the voiced speech of a particular person. Another example is the capture of information by medical test equipment, such as heart defibrilators. The features found in this pattern space would then be used to classify symptom sets into different disease categories.

In our classification example, the transducer and feature extractor of Figure 11.4 translates the problem information into parameters of a two-dimensional Cartesian space. Figure 11.5 presents the two-feature perceptron analysis of the information in Table 11.3. The first two columns of the table present the data points on which the network was trained. The third column represents the classification, +1 or −1, used as feedback in network training. Figure 11.5 is a graph of the training data of the problem, showing the linear separation of data classes created when the trained network was run on each data point.

We discuss first the general theory of classification. Each data grouping that a classifier identifies is represented by a region in multidimensional space. Each class R_i has a discriminant function g_i measuring membership in that region. Within the region R_i, the ith discriminant function has the largest value:

$$g_i(x) > g_j(x) \text{ for all } j,\ 1 < j < n.$$

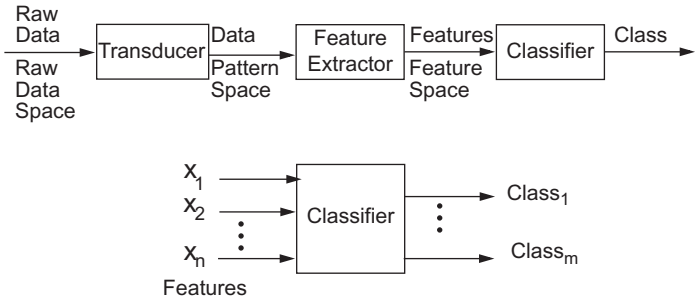


Figure 11.4 A full classification system.

In the simple example of Table 11.3, the two input parameters produce two obvious regions or classes in the space, one represented by 1, the other by –1.

x_1	x_2	Output
1.0	1.0	1
9.4	6.4	–1
2.5	2.1	1
8.0	7.7	–1
0.5	2.2	1
7.9	8.4	–1
7.0	7.0	–1
2.8	0.8	1
1.2	3.0	1
7.8	6.1	–1

Table 11.3 A data set for perceptron classification.

An important special case of discriminant functions is one which evaluates class membership based on the distance from some central point in the region. Classification based on this discriminant function is called *minimum distance classification*. A simple argument shows that if the classes are linearly separable there is a minimum distance classification.

If the regions of R_i and R_j are adjacent, as are the two regions in Figure 11.5, there is a boundary region where the discriminant functions are equal:

$g_i(x) = g_j(x)$ or $g_i(x) - g_j(x) = 0$.

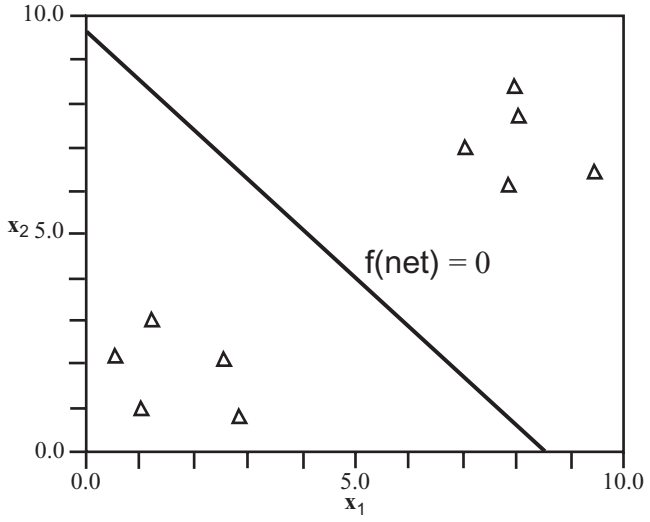


Figure 11.5 A two-dimensional plot of the data points in Table 11.3. The perceptron of Section 11.2.1 provides a linear separation of the data sets.

If the classes are linearly separable, as in Figure 11.5, the discriminant function separating the regions is a straight line, or $g_i(x) - g_j(x)$ is linear. Since a line is the locus of points equally distant from two fixed points, the discriminant functions, $g_i(x)$ and $g_j(x)$, are minimum distance functions, measured from the Cartesian center of each of the regions.

The perceptron of Figure 11.6 will compute this linear function. We need two input parameters and will have a bias with a constant value of 1. The perceptron computes:

$f(net) = f(w_1*x_1 + w_2*x_2 + w_3*1)$, where $f(x)$ is the sign of x.

When $f(x)$ is +1, x is interpreted as being in one class, when it is −1, x is in the other class. This thresholding to +1 or −1 is called linear bipolar thresholding (see Figure 11.7a). The bias serves to shift the thresholding function on the horizontal axis. The extent of this shift is learned by adjusting the weight w_3 during training.

We now use the data points of Table 11.3 to train the perceptron of Figure 11.6. We assume random initialization of the weights to [.75, .5, –.6] and use the perceptron training algorithm of Section 11.2.1. The superscripts, e.g., the 1 in $f(net)^1$, represent the current iteration number of the algorithm. We start by taking the first data point in the table:

$f(net)^1 = f(.75* 1 + .5*1 - .6*1) = f(.65) = 1$

Since $f(net)^1 = 1$, the correct output value, we do not adjust the weights. Thus $W^2 = W^1$. For our second data point:

$f(net)^2 = f(.75* 9.4 + .5*6.4 - .6*1) = f(9.65) = 1$

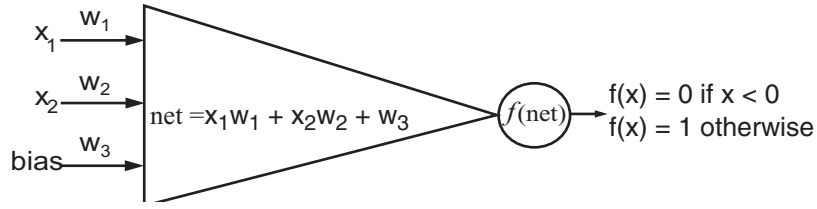


Figure 11.6 The perceptron net for the example data of Table 11.3. The thresholding function is linear and bipolar (see Figure 11.7a).

This time our result should have been –1 so we have to apply the learning rule, described in Section 11.1.1:

$$W^t = W^{t-1} + c(d^{t-1} - \text{sign}\,(W^{t-1} * X^{t-1}))\, X^{t-1}$$

where c is the learning constant, X and W are the input and weight vectors, and t the iteration of the net. d^{t-1} is the desired result at time $t - 1$, or in our situation, at $t = 2$. The net output at $t = 2$ is 1. Thus the difference between the desired and actual net output, $d^2 - \text{sign}\,(W^2 * X^2)$, is –2. In fact, in a hard limited bipolar perceptron, the learning increment will always be either +2c or else –2c times the training vector. We let the learning constant be a small positive real number, 0.2. We update the weight vector:

$$W^3 = W^2 + 0.2(-1-1)X^2 = \begin{bmatrix} 0.75 \\ 0.50 \\ -0.60 \end{bmatrix} - 0.4 \begin{bmatrix} 9.4 \\ 6.4 \\ 1.0 \end{bmatrix} = \begin{bmatrix} -3.01 \\ -2.06 \\ -1.00 \end{bmatrix}$$

We now consider the third data point with the newly adjusted weights:

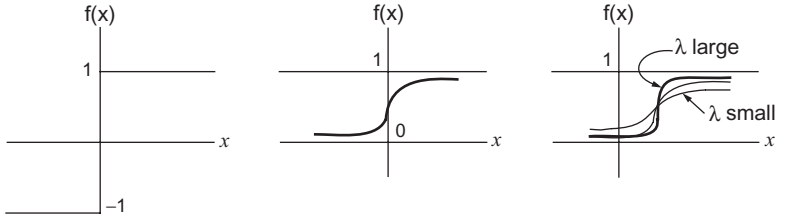
$$f(\text{net})^3 = f(-3.01 * 2.5 - 2.06 * 2.1 - 1.0 * 1) = f(-12.84) = -1$$

Again, the net result is not the desired output. We show the W^4 adjustment:

$$W^4 = W^3 + 0.2(1-(-1))X^3 = \begin{bmatrix} -3.01 \\ -2.06 \\ -1.00 \end{bmatrix} + 0.4 \begin{bmatrix} 2.5 \\ 2.1 \\ 1.0 \end{bmatrix} = \begin{bmatrix} -2.01 \\ -1.22 \\ -0.60 \end{bmatrix}$$

After 10 iterations of the perceptron net, the linear separation of Figure 11.5 is produced. After repeated training on the data set, about 500 iterations in total, the weight vector converges to [–1.3, –1.1, 10.9]. We are interested in the line separating the two classes. In terms of the discriminant functions g_i and g_j, the line is defined as the locus of points at which $g_i(x) = g_j(x)$ or $g_i(x) - g_j(x) = 0$, that is, where the net output is 0. The equation for the net output is given in terms of the weights. It is:

$$\text{output} = w_1x_1 + w_2x_2 + w_3.$$



a. A hard limiting and bipolar linear threshold.

b. A sigmoidal and unipolar threshold.

c. The sigmoidal, biased and squashed. As λ gets larger the sigmoid approximates a linear threshold.

Figure 11.7 Thresholding functions.

Consequently, the line separating the two classes is defined by the linear equation:

$$-1.3*x_1 + -1.1*x_2 + 10.9 = 0.$$

11.2.3 The Generalized Delta Rule

A straightforward way to generalize the perceptron network is to replace its hard limiting thresholding function with other types of activation functions. For example, continuous activation functions offer the possibility of more sophisticated learning algorithms by allowing for a finer granularity in error measurement.

Figure 11.7 shows the graph of some thresholding functions: a linear bipolar threshold function, Figure 11.7a, similar to that used by the perceptron, and a number of *sigmoidal* functions. Sigmoidal functions are so called because their graph is an "S"-shaped curve, as in Figure 11.7b. A common sigmoidal activation function, called the *logistic* function, is given by the equation:

$$f(net) = 1/(1 + e^{-\lambda * net}), \text{ where net } = \Sigma x_i w_i$$

As with previously defined functions, x_i is the input on line i, w_i is the weight on line i, and λ a "squashing parameter" used to fine-tune the sigmoidal curve. As λ gets large, the sigmoid approaches a linear threshold function over $\{0,1\}$; as it gets closer to 1 it approaches a straight line.

These threshold graphs plot the input values, the activation level of the neuron, against the scaled activation or output of the neuron. The sigmoidal activation function is continuous, which allows a more precise measure of error. Like the hard limiting thresholding function, the sigmoidal activation function maps most values in its domain into regions close to 0 or 1. However, there is a region of rapid but continuous transition between 0 and 1. In a sense, it approximates a thresholding behavior while providing a continuous output function. The use of λ in the exponent adjusts the slope of the sigmoid shape in the transition region. A weighted *bias* shifts the threshold along the x-axis.

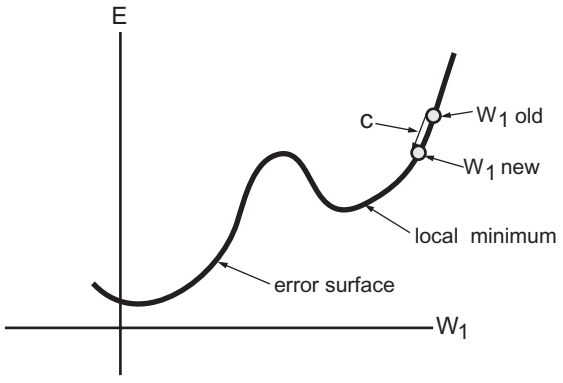


Figure 11.8 An error surface in two dimensions (more realistically must be visualized in n dimensions). Constant c is the size of the learning step.

The historical emergence of networks with continuous activation functions suggested new approaches to error reduction learning. The Widrow–Hoff (1960) learning rule is independent of the activation function, minimizing the squared error between the desired output value and the network activation, $net_i = WX_i$. Perhaps the most important learning rule for continuous activation functions is the *delta rule* (Rumelhart et al. 1986a).

Intuitively, the delta rule is based on the idea of an error surface, as illustrated in Figure 11.8. This error surface represents cumulative error over a data set as a function of network weights. Each possible network weight configuration is represented by a point on this n-dimensional error surface. Given a weight configuration, we want our learning algorithm to find the direction on this surface which most rapidly reduces the error. This approach is called *gradient descent learning* because the gradient is a measure of slope, as a function of direction, from a point on a surface.

To use the delta rule, the network must use an activation function which is continuous and therefore differentiable. The logistic formula just presented has this property. The delta rule learning formula for weight adjustment on the jth input to the ith node is:

$$c\,(d_i - O_i)\,f'\,(net_i)\,x_j,$$

where c is the constant controlling the learning rate, d_i and O_i are the desired and actual output values of the ith node. The derivative of the activation function for the ith node is f', and x_j is the jth input to node i. We now show the derivation of this formula.

The mean squared network error is found by summing the squared error for each node:

$$\text{Error} = (1/2)\sum_i (d_i - O_i)^2$$

where d_i is the desired value for each output node and O_i is the actual output of the node. We square each error so that the individual errors, some possibly with negative and others with positive values, will not, in summation, cancel each other out.

We consider here the case where the node is in the output layer; we describe the general case when we present networks with hidden layers in Section 11.3. We want first to measure the rate of change of network error with respect to output of each node. To do this we use the notion of a *partial derivative,* which gives us the rate of change of a multivariable function with respect to a particular variable. The partial derivative of the total error with respect to each output unit i is:

$$\frac{\partial \text{Error}}{\partial O_i} = \frac{\partial (1/2)*\Sigma(d_i - O_i)^2}{\partial O_i} = \frac{\partial (1/2)*(d_i - O_i)^2}{\partial O_i}$$

The second simplification is possible because we are considering a node on the output layer, where its error will not affect any other node. Taking the derivative of this quantity, we get:

$$\frac{\partial (1/2)*(d_i - O_i)^2}{\partial O_i} = -(d_i - O_i)$$

What we want is the rate of change of network error as a function of change in the weights at node i. To get the change in a particular weight, w_k, we rely on the use of the partial derivative, this time taking the partial derivative of the error at each node with respect to the weight, w_k, at that node. The expansion on the right side of the equal sign is given us by the chain rule for partial derivatives:

$$\frac{\partial \text{Error}}{\partial w_k} = \frac{\partial \text{Error}}{\partial O_i} * \frac{\partial O_i}{\partial w_k}$$

This gives us the pieces we need to solve the equation. Using our earlier result, we obtain:

$$\frac{\partial \text{Error}}{\partial w_k} = -(d_i - O_i) * \frac{\partial O_i}{\partial w_k}$$

We continue by considering the right most factor, the partial derivative of the actual output at the ith node taken with respect to each weight at that node. The formula for the output of node i as a function of its weights is:

$O_i = f(W_i X_i)$, where $W_i X_i = \text{net}_i$.

Since f is a continuous function, taking the derivative we get:

$$\frac{\partial O_i}{\partial w_k} = x_k * f'(W_i X_i) = f'(\text{net}_i) * x_k$$

Substituting in the previous equation:

$$\frac{\partial \text{Error}}{\partial w_k} = -(d_i - O_i) f'(\text{net}_i) * x_k$$

The minimization of the error requires that the weight changes be in the direction of the negative gradient component. Therefore:

$$\Delta w_k = -c\frac{\partial Error}{\partial w_k} = -c[-(d_i - O_i)*f'(net_i)*x_k] = c(d_i - O_i)f'(net_i)*x_k$$

We observe that the delta rule is like *hill-climbing,* Section 4.1, in that at every step, it attempts to minimize the local error measure by using the derivative to find the slope of the error space in the region local to a particular point. This makes delta learning vulnerable to the problem of distinguishing local from global minima in the error space.

The learning constant, c, exerts an important influence on the performance of the delta rule, as further analysis of Figure 11.8 illustrates. The value of c determines how much the weight values move in a single learning episode. The larger the value of c, the more quickly the weights move toward an optimal value. However, if c is too large, the algorithm may overshoot the minimum or oscillate around the optimal weights. Smaller values of c are less prone to this problem, but do not allow the system to learn as quickly. The optimal value of the learning rate, sometimes enhanced with a momentum factor (Zurada 1992), is a parameter adjusted for a particular application through experiment.

Although the delta rule does not by itself overcome the limitations of single layer networks, its generalized form is central to the functioning of the backpropagation algorithm, an algorithm for learning in a multilayer network. This algorithm is presented in the next section.

11.3 Backpropagation Learning

11.3.1 Deriving the Backpropagation Algorithm

As we have seen, single layer perceptron networks are limited as to the classifications that they can perform. We show in Sections 11.3 and 11.4 that the addition of multiple layers can overcome many of these limitations. In Section 16.3 we observe that multilayered networks are computationally complete, that is, equivalent to the class of Turing machines. Early researchers, however, were not able to design a learning algorithm for their use. We present in this section the generalized delta rule, which offers one solution to this problem.

The neurons in a multilayer network, as in Figure 11.9, are connected in layers, with units in layer n passing their activations only to neurons in layer n + 1. Multilayer signal processing means that errors deep in the network can spread and evolve in complex, unanticipated ways through successive layers. Thus, the analysis of the source of error at the output layer is complex. Backpropagation provides an algorithm for apportioning blame and adjusting weights accordingly.

The approach taken by the backpropagation algorithm is to start at the output layer and *propagate* error backwards through the hidden layers. When we analyzed learning with the delta rule, we saw that all the information needed to update the weights on a neuron was local to that neuron, except for the amount of error. For output nodes, this is easily computed as the difference between the desired and actual output values. For nodes in the

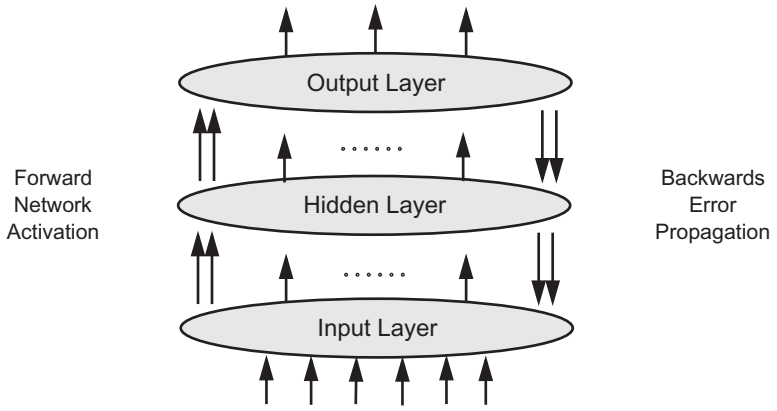


Figure 11.9 Backpropagation in a connectionist network having a hidden layer.

hidden layers, it is considerably more difficult to determine the error for which a node is responsible. The activation function for backpropagation is usually the logistic function:

$$f(net) = 1/(1 + e^{-\lambda * net}), \text{ where } net = \Sigma x_i w_i.$$

This function is used for four reasons: First, it has the sigmoid shape. Second, as a continuous function, it has a derivative everywhere. Third, since the value of the derivative is greatest where the sigmoidal function is changing most rapidly, the assignment of the most error is attributed to those nodes whose activation was least certain. Finally, the derivative is easily computed by a subtraction and multiplication:

$$f(net) = (1/(1 + e^{-\lambda * net})) = \lambda(f(net) * (1 - f(net))).$$

Backpropagation training uses the generalized delta rule. This uses the same gradient descent approach presented in Section 11.2. For nodes in the hidden layer we look at their contribution to the error at the output layer. The formulas for computing the adjustment of the weight w_{ki} on the path from the kth to the ith node in backpropagation training are:

1) $\Delta w_{ki} = -c(d_i - O_i) * O_i (1 - O_i) x_k$, for nodes on the output layer, and

2) $\Delta w_{ki} = -c * O_i (1 - O_i) \sum_j (-delta_j * w_{ij}) x_k$, for nodes on hidden layers.

In 2), j is the index of the nodes in the next layer to which i's signals fan out and:

$$delta_j = -\frac{\partial Error}{\partial net_j} = (d_i - O_i) * O_i (1 - O_i).$$

We now show the derivation of these formulae. First we derive 1), the formula for weight adjustment on nodes in the output layer. As before, what we want is the rate of change of network error as a function of change in the kth weight, w_k, of node i. We treated this situation in the derivation of the delta rule, Section 11.2.3, and showed that:

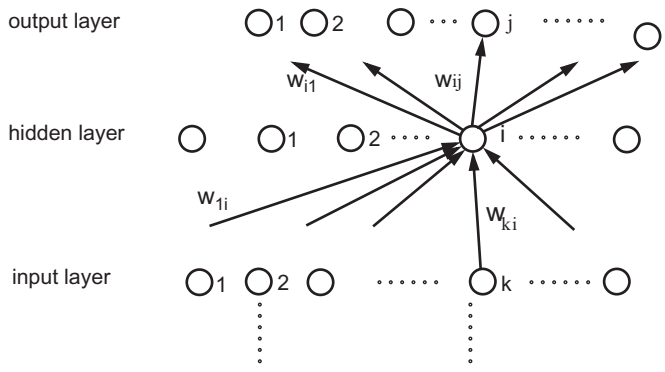


Figure 11.10 $\sum_j -delta_j * w_{ij}$ is the total contribution of node i to the error at the output. Our derivation gives the adjustment for w_{ki}.

$$\frac{\partial Error}{\partial w_k} = -((d_i - O_i) * f'(net_i) * x_k)$$

Since f, which could be any function, is now the logistic activation function, we have:

$$f'(net) = f'(1/(1 + e^{-\lambda * net})) = f(net) * (1 - f(net)).$$

Recall that $f(net_i)$ is simply O_i. Substituting in the previous equation, we get:

$$\frac{\partial Error}{\partial w_k} = -(d_i - O_i) * O_i * (1 - O_i) * x_k$$

Since the minimization of the error requires that the weight changes be in the direction of the negative gradient component, we multiply by –c to get the weight adjustment for the ith node of the output layer:

$$\Delta w_k = c(d_i - O_i) * O_i * (1 - O_i) * x_k.$$

We next derive the weight adjustment for hidden nodes. For the sake of clarity we initially assume a single hidden layer. We take a single node i on the hidden layer and analyze its contribution to the total network error. We do this by initially considering node i's contribution to the error at a node j on the output layer. We then sum these contributions across all nodes on the output layer. Finally, we describe the contribution of the kth input weight on node i to the network error. Figure 11.10 illustrates this situation.

We first look at the partial derivative of the network error with respect to the output of node i on the hidden layer. We get this by applying the chain rule:

$$\frac{\partial Error}{\partial O_i} = \frac{\partial Error}{\partial net_j} * \frac{\partial net_j}{\partial O_i}$$

The negative of the first term on the right-hand side, $(\delta Error) / (\delta net_j)$, is called $delta_j$. Therefore, we can rewrite the equation as:

$$\frac{\partial Error}{\partial O_i} = -delta_j * \frac{\partial net_j}{\partial O_i}$$

Recall that the activation of node j, net_j, on the output layer is given by the sum of the product of its own weights and of the output values coming from the nodes on the hidden layer:

$$net_j = \sum_i w_{ij} O_i$$

Since we are taking the partial derivative with respect to only one component of the sum, namely the connection between node i and node j, we get:

$$\frac{\partial net_j}{\partial O_i} = w_{ij},$$

where w_{ij} is the weight on the connection from node i in the hidden layer to node j in the output layer. Substituting this result:

$$\frac{\partial Error}{\partial O_i} = -delta_j * w_{ij}$$

Now we sum over all the connections of node i to the output layer:

$$\frac{\partial Error}{\partial O_i} = \sum_j - delta_j * w_{ij}$$

This gives us the sensitivity of network error to the output of node i on the hidden layer. We next determine the value of $delta_i$, the sensitivity of network error to the net activation at hidden node i. This gives the sensitivity of network error to the incoming weights of node i. Using the chain rule again:

$$-delta_i = \frac{\partial Error}{\partial net_i} = \frac{\partial Error}{\partial O_i} * \frac{\partial O_i}{\partial net_i}$$

Since we are using the logistic activation function,

$$\frac{\partial O_i}{\partial net_i} = O_i * (1 - O_i)$$

We now substitute this value in the equation for $delta_i$ to get:

$$-delta_i = O_i * (1 - O_i) * \sum_j -delta_j * w_{ij}$$

Finally, we can evaluate the sensitivity of the network error on the output layer to the incoming weights on hidden node i. We examine the kth weight on node i, w_k. By the chain rule:

$$\frac{\partial Error}{\partial w_{ki}} = \frac{\partial Error}{\partial net_i} * \frac{\partial net_i}{\partial w_{ki}} = -delta_i * \frac{\partial net_i}{\partial w_{ki}} = -delta_i * x_k$$

where x_k is the kth input to node i.

We substitute into the equation the value of $-delta_i$:

$$\frac{\partial Error}{\partial w_{ki}} = O_i(1 - O_i)\sum_j(-delta_j * w_{ij})x_k$$

Since the minimization of the error requires that the weight changes be in the direction of the negative gradient component, we get the weight adjustment for the kth weight of i by multiplying by the negative of the learning constant:

$$\Delta w_{ki} = -c\frac{\partial Error}{\partial w_{ki}} = c * O_i(1 - O_i)\sum_j(delta_j * w_{ij})x_k \ .$$

For networks with more than one hidden layer, the same procedure is applied recursively to propagate the error from hidden layer n to hidden layer n – 1.

Although it provides a solution to the problem of learning in multilayer networks, backpropagation is not without its own difficulties. As with hillclimbing, it may converge to local minima, as in Figure 11.8. Finally, backpropagation can be expensive to compute, especially when the network converges slowly.

11.3.2 Backpropagation Example 1: NETtalk

NETtalk is an interesting example of a neural net solution to a difficult learning problem (Sejnowski and Rosenberg 1987). NETtalk learned to pronounce English text. This can be a difficult task for an explicit symbol approach, for example a rule-based system, since English pronunciation is highly irregular.

NETtalk learned to read a string of text and return a phoneme and an associated stress for each letter in the string. A phoneme is the basic unit of sound in a language; the stress is the relative loudness of that sound. Because the pronunciation of a single letter depends upon its context and the letters around it, NETtalk was given a seven character window. As the text moves through this window, NETtalk returns a phoneme/stress pair for each letter.

Figure 11.11 shows the architecture of NETtalk. The network consists of three layers of units. The input units correspond to the seven character window on the text. Each position in the window is represented by 29 input units, one for each letter of the alphabet, and 3 for punctuation and spaces. The letter in each position activates the corresponding unit. The output units encode phonemes using 21 different features of human articulation. The remaining five units encoded stress and syllable boundaries. NETtalk has 80 hidden units, 26 output values, and 18,629 connections.

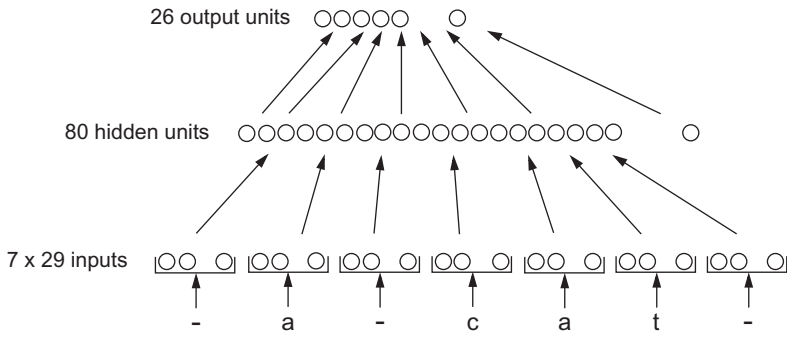


Figure 11.11 The network topology of NETtalk.

NETtalk is trained by giving it a seven character window and letting it attempt to pronounce the middle character. Comparing its attempted pronunciation to the correct pronunciation, it adjusts its weights using backpropagation.

This example illustrates a number of interesting properties of neural networks, many of which reflect the nature of human learning. For example, learning, when measured as a percentage of correct responses, proceeds rapidly at first, and slows as the percentage correct increases. As with humans, the more words the network learns to pronounce, the better it is at correctly pronouncing new words. Experiments in which some of the weights in a fully trained network were randomly altered showed the network to be damage resistant, degrading gracefully as weights were altered. Researchers also found that relearning in a damaged network was highly efficient.

Another interesting aspect of multilayered networks is the role of the hidden layers. Any learning algorithm must learn generalizations that apply to unseen instances in the problem domain. The hidden layers play an important role in allowing a neural network to generalize. NETtalk, like many backpropagation networks, has fewer neurons in the hidden layer than in the input layer. This means that since fewer nodes on the hidden layer are used to encode the information in the training patterns, some form of abstraction is taking place. The shorter encoding implies that different patterns on the input layer can be mapped into identical patterns at the hidden layer. This reduction is a generalization.

NETtalk learns effectively, although it requires a large number of training instances, as well as repeated passes through the training data. In a series of empirical tests comparing backpropagation and ID3 on this problem, Shavlik et al. (1991) found that the algorithms had equivalent results, although their training and use of data was quite different. This research evaluated the algorithms by dividing the total set of examples into separate training and test sets. Both ID3 (Section 9.3) and NETtalk were able to correctly pronounce about 60 per cent of the test data after training on 500 examples. But, where ID3 required only a single pass through the training data, NETtalk required many repetitions of the training set. In this research, NETtalk had 100 passes through the training data.

As our example demonstrates, the relationship between connectionist and symbolic learning is more complicated than it might seem at first. In our next example we work through the details of a backpropagation solution to the exclusive-or problem.

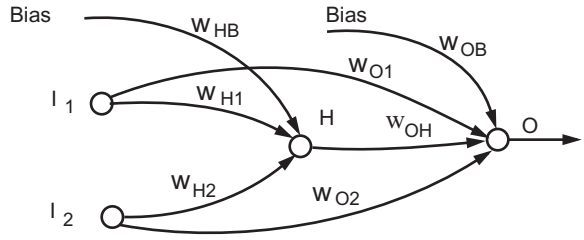


Figure 11.12 A backpropagation net to solve the exclusive-or problem. The W_{ij} are the weights and H is the hidden node.

11.3.3 Backpropagation Example 2: Exclusive-or

We end this section by presenting a simple hidden layer solution to the *exclusive-or* problem. Figure 11.12 shows a network with two input nodes, one hidden node and one output node. The network also has two bias nodes, the first to the hidden node and the second to the output node. The net values for the hidden and output nodes are calculated in the usual manner, as the vector product of the input values times their trained weights. The bias is added to this sum. The weights are trained by backpropagation and the activation function is sigmoidal.

It should be noted that the input nodes are also directly linked, with trained weights, to the output node. This additional linking can often let the designer get a network with fewer nodes on the hidden layer and quicker convergence. In fact there is nothing unique about the network of Figure 11.12; any number of different networks could be used to compute exclusive-or.

We trained our randomly initialized network with multiple instances of the four patterns that represent the truth values of exclusive-or:

$$(0, 0) \rightarrow 0;\ (1, 0) \rightarrow 1;\ (0, 1) \rightarrow 1;\ (1, 1) \rightarrow 0$$

A total of 1400 training cycles using these four instances produced the following values, rounded to the nearest tenth, for the weight parameters of Figure 11.12:

$$W_{H1} = -7.0 \quad W_{HB} = 2.6 \quad W_{O1} = -5.0 \quad W_{OH} = -11.0$$
$$W_{H2} = -7.0 \quad W_{OB} = 7.0 \quad W_{O2} = -4.0$$

With input values (0, 0), the output of the hidden node is:

$$f(0*(-7.0) + 0*(-7.0) + 1*2.6) = f(2.6) \rightarrow 1$$

The output of the output node for (0,0) is:

$$f(0*(-5.0) + 0*(-4.0) + 1*(-11.0) + 1*(7.0)) = f(-4.0) \rightarrow 0$$

With input values (1, 0), the output of the hidden node is:

$$f(1*(-7.0) + 0*(-7.0) + 1*2.6) = f(-4.4) \rightarrow 0$$

The output of the output node for (1,0) is:

$$f(1*(-5.0) + 0*(-4.0) + 0*(-11.0) + 1*(7.0)) = f(2.0) \rightarrow 1$$

The input value of (0, 1) is similar. Finally, let us check our exclusive-or network with input values of (1, 1). The output of the hidden node is:

$$f(1*(-7.0) + 1*(-7.0) + 1*2.6) = f(-11.4) \rightarrow 0$$

The output of the output node for (1,1) is:

$$f(1*(-5.0) + 1*(-4.0) + 0*(-11.0) + 1*(7.0)) = f(-2.0) \rightarrow 0$$

The reader can see that this feedforward network with backpropagation learning made a nonlinear separation of these data points. The threshold function f is the sigmoidal of Figure 11.7b, the learned biases have translated it very slightly in the positive direction on the x-axis.

We next consider models of competitive learning.

11.4 Competitive Learning

11.4.1 Winner-Take-All Learning for Classification

The winner-take-all algorithm (Kohonen 1984, Hecht-Nielsen 1987) works with the single node in a layer of nodes that responds most strongly to the input pattern. Winner-take-all may be viewed as a competition among a set of network nodes, as in Figure 11.13. In this figure we have a vector of input values, $X=(x_1, x_2, ..., x_m)$, passed into a layer of network nodes, A, B,..., N. The diagram shows node B the winner of the competition, with an output signal of 1.

Learning for winner-take-all is unsupervised in that the winner is determined by a "maximum activation" test. The weight vector of the winner is then rewarded by bringing its components closer to those of the input vector. For the weights, W, of the winning node and components X of the input vector, the increment is:

$$\Delta W^t = c(X^{t-1} - W^{t-1}),$$

where c is a small positive learning constant that usually decreases as the learning proceeds. The winning weight vector is then adjusted by adding ΔW^t.

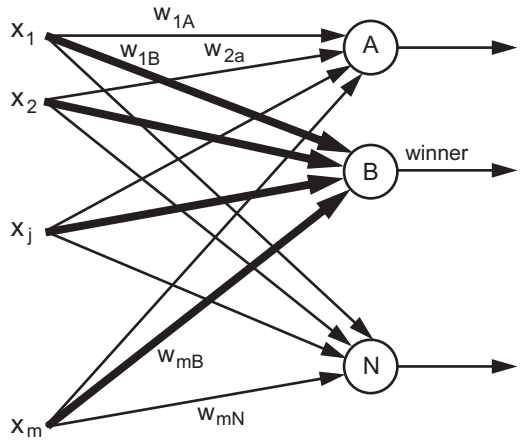


Figure 11.13 A layer of nodes for application of a winner-take-all algorithm. The old input vectors support the winning node.

This reward increments or decrements each component of the winner's weight vector by a fraction of the $x_i - w_i$ difference. The effect is, of course, to make the winning node match more closely the input vector. The winner-take-all algorithm does not need to directly compute activation levels to find the node with the strongest response. The activation level of a node is directly related to the closeness of its weight vector to the input vector. For a node i with a normalized weight vector W_i, the activation level, W_iX, is a function of the Euclidean distance between W_i and the input pattern X. This can be seen by calculating the Euclidean distance, with normalized W_i:

$$\|X - W_i\| = \sqrt{(X - W_i)^2} = \sqrt{X^2 - 2XW_i + W_i^2}$$

From this equation it can be seen that for a set of normalized weight vectors, the weight vector with the smallest Euclidean distance, $\| X - W \|$, will be the weight vector with the maximum activation value, WX. In many cases it is more efficient to determine the winner by calculating Euclidean distances rather than comparing activation levels on normalized weight vectors.

We consider the "winner-take-all" Kohonen learning rule for several reasons. First, we consider it as a classification method and compare it to perceptron classification. Second, it may be combined with other network architectures to offer more sophisticated models of learning. We look at the combination of Kohonen prototype learning with an outstar, supervised learning network. This hybrid, first proposed by Robert Hecht-Nielsen (1987, 1990), is called a *counterpropagation* network. We see, in Section 11.4.3, how we can describe conditioned learning using counterpropagation.

Before we leave this introduction, there are a number of issues important for "winner-take-all" algorithms. Sometimes a "conscience" parameter is set and updated at each iteration to keep individual nodes from winning too often. This ensures that all network nodes

eventually participate in representing the pattern space. In some algorithms, rather than identifying a winner that takes all, a *set* of closest nodes are selected and the weights of each are differentially incremented. Another approach is to differentially reward the neighboring nodes of the winner. Weights are typically initialized at random values and then normalized during this learning method (Zurada 1992). Hecht-Nielsen (1990) shows how "winner-take-all" algorithms may be seen as equivalent to the k-means analysis of a set of data. In the next section we present Kohonen's winner-take-all unsupervised method for the learning of clusters.

11.4.2 A Kohonen Network for Learning Prototypes

Classification of data and the role of prototypes in learning are constant concerns of psychologists, linguists, computer scientists, and cognitive scientists (Wittgenstein 1953, Rosch 1978, Lakoff 1987). The role of prototypes and classification in intelligence is also a constant theme of this book. We demonstrated symbol based-classification and probabilistic clustering algorithms with COBWEB and CLUSTER/2 in Section 9.5. In connectionist models, we demonstrated perceptron-based classification in Section 11.2 and now show a Kohonen (1984) winner-take-all clustering algorithm.

Figure 11.14 presents again the data points of Table 11.3. Superimposed on these points are a series of prototypes created during network training. The perceptron training algorithm converged after a number of iterations, resulting in a network weight configuration defining a linear separation between the two classes. As we saw, the line defined by these weights was obtained by implicitly computing the Euclidean "center" of each cluster. This center of a cluster serves in perceptron classification as a prototype of the class.

Kohonen learning, on the other hand, is unsupervised, with a set of prototypes randomly created and then refined until they come to explicitly represent the clusters of data. As the algorithm continues, the learning constant is progressively reduced so that each new input vector will cause less perturbation in the prototypes.

Kohonen learning, like CLUSTER/2, has a strong inductive bias in that the number of desired prototypes is explicitly identified at the beginning of the algorithm and then continuously refined. This allows the net algorithm designer to identify a specific number of prototypes to represent the clusters of data. Counterpropagation (Section 11.4.3) allows further manipulation of this selected number of prototypes.

Figure 11.15 is a Kohonen learning network for classification of the data of Table 11.3. The data are represented in Cartesian two dimensional space, so prototypes to represent the data clusters will also be ordered pairs. We select two prototypes, one to represent each data cluster. We have randomly initialized node A to (7, 2) and node B to (2, 9). Random initialization only works in simple problems such as ours; an alternative is to set the weight vectors equal to representatives of each of the clusters.

The winning node will have a weight vector closest to that of the input vector. This weight vector for the winning node will be rewarded by being moved even closer to the input data, while the weights on the losing nodes are left unchanged. Since we are explicitly calculating the Euclidean distance of the input vector from each of the prototypes we will not need to normalize the vectors, as described in Section 11.4.1.

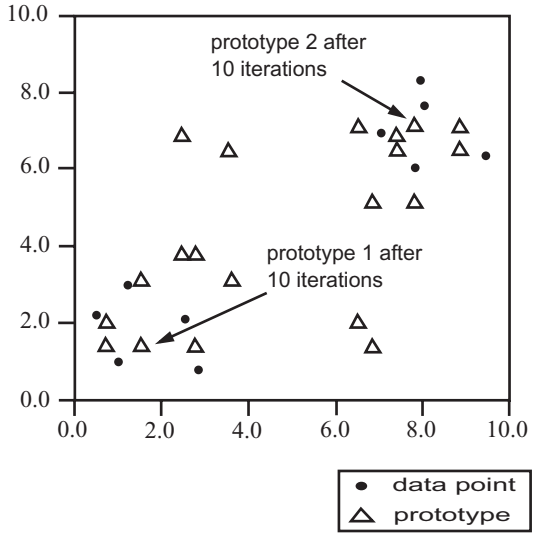


Figure 11.14 The use of a Kohonen layer, unsupervised, to generate a sequence of prototypes to represent the classes of Table 11.3.

Kohonen learning is unsupervised, in that a simple measure of the distance between each prototype and the data point allows selection of the winner. Classification will be "discovered" in the context of this *self-organizing* network. Although Kohonen learning selects data points for analysis in random order, we take the points of Table 11.3 in top to bottom order. For point (1, 1), we measure the distance from each prototype:

$||(1, 1) – (7, 2)|| = (1 – 7)^2 + (1 – 2)^2 = 37$, and
$||(1, 1) – (2, 9)|| = (1 – 2)^2 + (1 – 9)^2 = 65$.

Node A (7, 2) is the winner since it is closest to (1, 1). $||(1, 1) – (7, 2)||$ represents the distance between these two points; we do not need to apply the square root function in the Euclidean distance measure because the relation of magnitudes is invariant. We now reward the winning node, using the learning constant c set to 0.5. For the second iteration:

$$W^2 = W^1 + c(X^1 – W^1)$$
$$= (7, 2) + .5((1, 1) – (7, 2)) = (7, 2) + .5((1 – 7), (1 – 2))$$
$$= (7, 2) + (–3, –.5) = (4, 1.5)$$

At the second iteration of the learning algorithm we have, for data point (9.4, 6.4):

$||(9.4, 6.4) – (4, 1.5)|| = (9.4 – 4)^2 + (6.4 – 1.5)^2 = 53.17$ and
$||(9.4, 6.4) – (2, 9)|| = (9.4 – 2)^2 + (6.4 – 9)^2 = 60.15$

Again, node A is the winner. The weight for the third iteration is:

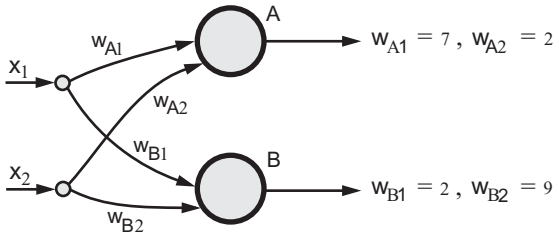


Figure 11.15 The architecture of the Kohonen based learning network for the data of Table 11.3 and classification of Figure 11.14.

$$W^3 = W^2 + c(X^2 - W^2)$$
$$= (4, 1.5) + .5((9.4, 6.4) - (4, 1.5))$$
$$= (4, 1.5) + (2.7, 2.5) = (6.7, 4)$$

At the third iteration we have, for data point (2.5, 2.1):

$$||(2.5, 2.1) - (6.7, 4)|| = (2.5 - 6.7)^2 + (2.1 - 4)^2 = 21.25, \text{ and}$$
$$||(2.5, 2.1) - (2, 9)|| = (2.5 - 2)^2 + (2.1 - 9)^2 = 47.86.$$

Node A wins again and we go on to calculate its new weight vector. Figure 11.14 shows the evolution of the prototype after 10 iterations. The algorithm used to generate the data of Figure 11.14 selected data randomly from Table 11.3, so the prototypes shown will differ from those just created. The progressive improvement of the prototypes can be seen moving toward the centers of the data clusters. Again, this is an unsupervised, winner-take-all reinforcement algorithm. It builds a set of evolving and explicit prototypes to represent the data clusters. A number of researchers, including Zurada (1992) and Hecht-Nielsen (1990), point out that Kohonen unsupervised classification of data is basically the same as k-means analysis.

We next consider, with a Grossberg, or outstar, extension of Kohonen winner-take-all analysis, an algorithm that will let us extend the power of prototype selection.

11.4.3 Outstar Networks and Counterpropagation

To this point we considered the unsupervised clustering of input data. Learning here requires little *a priori* knowledge of a problem domain. Gradually detected characteristics of the data, as well as the training history, lead to the identification of classes and the discovery of boundaries between them. Once data points are clustered according to similarities in their vector representations, a teacher can assist in calibrating or giving names to data classes. This is done by a form of supervised training, where we take the output nodes of a "winner-take-all" network layer and use them as input to a second network layer. We will then explicitly reinforce decisions at this output layer.

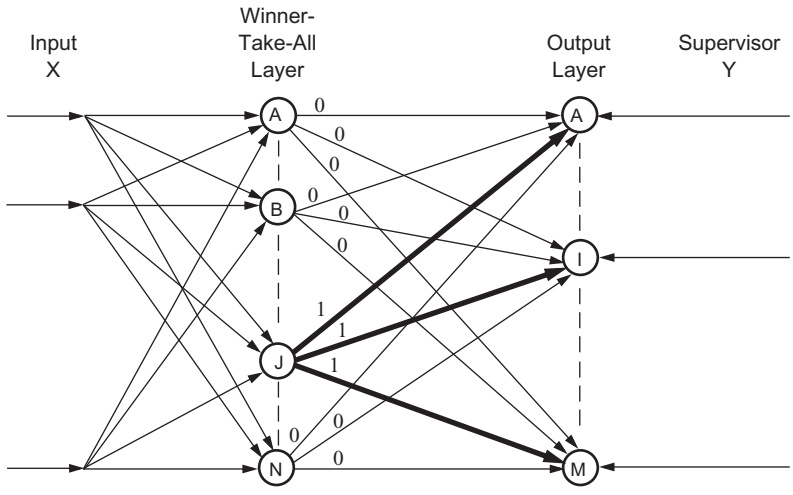


Figure 11.16 The outstar of node J, the winner in a winner-take-all net work. The Y vector supervises the response on the output layer in Grossberg training. The outstar is bold with all weights 1; all other weights are 0.

This supervised training and then reinforced output allows us, for example, to map the results of a Kohonen net into an output pattern or class. A Grossberg (1982, 1988) layer, implementing an algorithm called *outstar*, allows us to do this. The combined network, a Kohonen layer joined to a Grossberg layer, is called *counterpropagation* and was first proposed by Robert Hecht-Nielsen (1987, 1990).

In Section 11.4.2 we considered in some detail the Kohonen layer; here we consider the Grossberg layer. Figure 11.16 shows a layer of nodes, A, B, ..., N, where one node, J, is selected as the winner. Grossberg learning is supervised in that we wish, with feedback from a teacher, represented by vector Y, to reinforce the weight connecting J to the node I in the output layer which is supposed to fire. With outstar learning, we identify and increase the weight w_{JI} on the outbound link of J to I.

To train the counterpropagation net we first train the Kohonen layer. When a winner is found, the values on all the links going out from it will be 1, while all the output values of its competitors remain 0. That node, together with all the nodes on the output layer to which it is connected, form what is called an *outstar* (see Figure 11.16). Training for the Grossberg layer is based on outstar components.

If each cluster of input vectors represents a single class and we want all members of a class to map onto the same value at the output layer, we do not need an iterative training. We need only determine which node in the winner-take-all layer is linked to which class and then assign weights from those nodes to output nodes based on the association between classes and desired output values. For example, if the Jth winner-take-all unit wins for all elements of the cluster for which $I = 1$ is the desired output of the network, we set $w_{JI} = 1$ and $w_{JK} = 0$ for all other weights on the outstar of J.

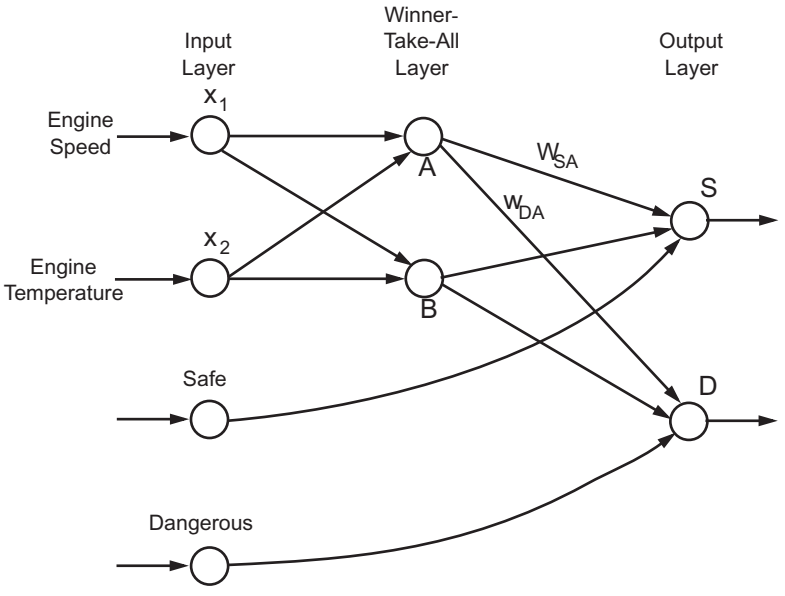


Figure 11.17 A counterpropagation network to recognize the classes in Table 11.3. We train the outstar weights of node A, w_{SA} and w_{DA}.

If the desired output for elements of a cluster vary, then there is an iterative procedure, using the supervision vector Y, for adjusting outstar weights. The result of this training procedure is to *average* the desired output values for elements of a particular cluster. We train the weights on the outstar connections from the winning node to the output nodes according to the equation:

$$W^{t+1} = W^t + c(Y - W^t)$$

where c is a small positive learning constant, W^t is the weight vector of the outstar component, and Y is the desired output vector. Note that this learning algorithm has the effect of increasing the connection between node J on the Kohonen layer and node I on the output layer precisely when I is a winning node with an output of 1 and the desired output of J is also 1. This makes it an instance of Hebbian learning, a form of learning in which a neural pathway is strengthened every time one node contributes to the firing of another. We discuss Hebbian learning in more detail in Section 11.5.

We next apply the rule for training a counterpropagation network to recognize the data clusters of Table 11.3. We also show with this example how counterpropagation nets implement conditioned learning. Suppose the x_1 parameter in Table 11.3 represents engine speed in a propulsion system. x_2 represents engine temperature. Both the speed and the temperature of the system are calibrated to produce data points in the range [0, 10]. Our monitoring system samples data points at regular intervals. Whenever speed and temperature are excessively high, we want to broadcast a warning. Let us rename the output values

of Table 11.3 from +1 to "safe" and from –1 to "dangerous." Our counterpropagation network will look like Figure 11.17.

Since we know exactly what values we want each winning node of the Kohonen net to map to on the output layer of the Grossberg net, we could directly set those values. To demonstrate outstar learning, however, we will train the net using the formula just given. If we make the (arbitrary) decision that node S on the output layer should signal safe situations and node D dangerous, then the outstar weights for node A on the output layer of the Kohonen net should be [1, 0] and the outstar weights for B should be [0, 1]. Because of the symmetry of the situation, we show the training of the outstar for node A only.

The Kohonen net must have stabilized before the Grossberg net can be trained. We demonstrated the Kohonen convergence of this same net in Section 11.4.2. The input vectors for training the A outstar node are of the form $[x_1, x_2, 1, 0]$. x_1 and x_2 are values from Table 11.3 that are clustered at Kohonen output node A and the last two components indicate that when A is the Kohonen winner, safe is "true" and dangerous is "false," as in Figure 11.15. We initialize the outstar weights of A to [0, 0] and use .2 as the learning constant:

$W^1 = [0, 0] + .2[[1, 0] – [0, 0]] = [0, 0] + [.2, 0] = [.2, 0]$
$W^2 = [.2, 0] + .2[[1, 0] – [.2, 0]] = [.2, 0] + [.16, 0] = [.36, 0]$
$W^3 = [.36, 0] + .2[[1, 0] – [.36, 0]] = [.36, 0] + [.13, 0] = [.49, 0]$
$W^4 = [.49, 0] + .2[[1, 0] – [.49, 0]] = [.49, 0] + [.10, 0] = [.59, 0]$
$W^5 = [.59, 0] + .2[[1, 0] – [.59, 0]] = [.59, 0] + [.08, 0] = [.67, 0].$

As we can see, with training these weights are moving toward [1, 0]. Of course, since in this case elements of the cluster associated with A always map into [1,0], we could have used the simple assignment algorithm rather than used the averaging algorithm for training.

We now show that this assignment gives the appropriate response from the counterpropagation net. When the first input vector from Table 11.3 is applied to the network in Figure 11.17, we get activation of [1, 1] for the outstar weights of node A and [0, 0] for the outstar of B. The dot product of activation and weights for node S of the output layer is [1, 0] * [1, 0]; this gives activation 1 to the S output node. With outstar weights of B trained to [0, 1], the activation for node D is [1, 0] * [0, 1]; these are the values that we expect. Testing the second row of data points on Table 11.3, we get activation [0, 0] from the A node and [1,1] from the B at the winner-take-all level. The dot product of these values and the trained weights gives 0 to the S node and 1 to D, again what is expected. The reader may continue to test other data from Table 11.3.

From a cognitive perspective, we can give an associationist interpretation to the counterpropagation net. Consider again Figure 11.17. The learning on the Kohonen layer can be seen as acquiring a conditioned stimulus, since the network is learning patterns in events. The learning on the Grossberg level, on the other hand, is an association of nodes (unconditioned stimuli) to some response. In our situation the system learns to broadcast a danger warning when data fit into a certain pattern. Once the appropriate response is learned, then even without the continued coaching of a teacher, the system responds appropriately to new data.

A second cognitive interpretation of counterpropagation is as the reinforcement of memory links for pattern of phenomena. This is similar to building a lookup table for responses to data patterns.

Counterpropagation has, in certain cases, a considerable advantage over backpropagation. Like backpropagation it is capable of learning nonlinearly separable classifications. It does this, however, by virtue of the preprocessing which goes on in the Kohonen layer, where the data set is partitioned into clusters of homogenous data. This partitioning can result in a significant advantage over backpropagation in learning rate since the explicit partitioning of data into separate clusters replaces the often extensive search required on the hidden layers in backpropagation networks.

11.4.4 Support Vector Machines (Harrison and Luger 2002)

Support vector machines (SVM), offer another example of competitive learning. In the support vector approach, statistical measures are used to determine a minimum set of data points (the support vectors) that maximally separate the positive and negative instances of a learned concept. These support vectors, representing selected data points from both the positive and negative instances of the concept, implicitly define a hyperplane separating these two data sets. For example, running the SVM algorithm identifies points (2.5, 2.1) and (1.2, 3.0) as support vectors for the positive instances and (7.0, 7.0) and (7.8. 6.1) as support vectors for the negative instances of the data of Table 11.3 and Figure 11.5. Once the support vectors are learned other data points need no longer be retained, the support vectors alone are sufficient to determine the separating hyperplane.

The support vector machine is a linear classifier where the learning of the support vectors is supervised. The data for SVM learning is assumed to be produced independently and identically from a fixed, although unknown, distribution of data. The hyperplane, implicitly defined by the support vectors themselves, divides the positive from the negative data instances. Data points nearest the hyperplane are in the *decision margin* (Burges 1998). Any addition or removal of a support vector changes the hyperplane boundary. As previously noted, after training is complete, it is possible to reconstruct the hyperplane and classify new data sets from the support vectors alone.

The SVM algorithm classifies data elements by computing the distance of a data point from the separating hyperplane as an optimization problem. Successfully controlling the increased flexibility of feature spaces, the (often transformed) parameters of the instances to be learned requires a sophisticated theory of generalization. This theory must be able to precisely describe the features that have to be controlled to form a good generalization. Within statistics, this issue is known as the *study of the rates of uniform convergence*. We have already seen an example of this in Section 10.4.2, where the probably approximately correct, or PAC model of learning is presented. The results of PAC learning can be seen as establishing bounds on the number of examples required to guarantee a specific error bound. For this generalization task Bayesian or other data compression techniques are employed. In SVM learning the theory of Vapnik and Chervonenkis is often used.

The Vapnik Chervonenkis (VC) dimension is defined as the maximum number of training points that can be divided into two categories by a set of functions (Burges 1998).

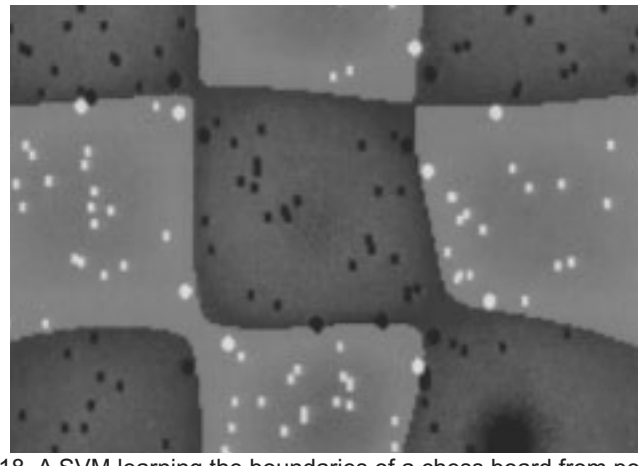


Figure 11.18 A SVM learning the boundaries of a chess board from points generated according to the uniform distribution using Gaussian kernels. The dots are the data points with the larger dots comprising the set of support vectors, the darker areas indicate the confidence in the classification. Adapted from (Cristianini and Shawe-Taylor 2000).

Thus VC theory provides a distribution free bound on the generalization of the consistent hypothesis (Cristianini and Shawe-Taylor 2000). The SVM algorithm uses this VC theory to compute the hyperplane and controls the margin of error for the generalizations accuracy, sometimes called the *capacity* of the function.

SVMs use a dot product similarity measure to map data from a feature space. Dot product results representing mapped vectors are linearly combined by weights found by solving a quadratic program (Scholkopf et al. 1998). A kernel function, such as a polynomial, spline, or Gaussian, is used to create the feature vector mapping, where kernel choice is determined by the problem distribution. SVMs compute distances to determine data element classification. These decision rules created by the SVM represent statistical regularities in the data. Once the SVM is trained, classification of new data points is simply a matter of comparison with the support vectors. In the support vectors, critical features characterizing the learned concept are clustered on one side of the hyperplane, those describing its negation on the other, and features that don't discriminate aren't used.

For the perceptron algorithm of Section 11.2, the linear separability of data is important: if the data is not separable the algorithm will not converge. The SVM, alternatively, attempts to maximize the decision margin and is more robust in its ability to handle poor separation caused by overlapping data points. It is able to use *slack* variables to relax the linear constraints to find a soft margin, with values that denote the confidence level of the classification boundary (Cristianini and Shawe-Taylor 2000). As a result some support vectors that are outliers may be misclassified to produce the hyperplane and as a result the decision margin will be narrowed when the data is noisy.

SVMs may be generalized from two category classification problems to the discrimination of multiple classes by repeatedly running the SVM on each category of interest

against all the other categories. SVMs are best suited to problems with numerical data rather than categorical; as a result their applicability for many classic categorization problems with qualitative boundaries is limited. Their strength lies in their mathematical foundations: minimization of a convex quadratic function under linear inequality constraints.

SVMs are applied to many learning situations, including the classification of web pages. In text categorization, the presence of search or other related words are weighted. Each document then becomes input data for the SVM to categorize on the basis of word frequency information (Harrison and Luger 2002). SVMs are also used for image recognition, focusing on edge detection and shape description using gray scale or color intensity information (Cristianini and Shawe-Taylor 2000). In Figure 11.18, adapted from Cristianini and Shawe-Taylor (2000), the support vector machine discriminates boundaries on a chess board. More details on SVMs and the full SVM learning algorithm may be found in (Burges 1998).

11.5 Hebbian Coincidence Learning

11.5.1 Introduction

Hebb's theory of learning is based on the observation that in biological systems when one neuron contributes to the firing of another neuron, the connection or pathway between the two neurons is strengthened. Hebb (1949) stated:

> When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes place in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.

Hebbian learning is appealing because it establishes behavior-based reward concepts on the neuronal level. Neural physiological research has confirmed that Hebb's idea that temporal proximity of the firing of connected neurons can modify synaptic strength, albeit in a much more complex fashion that Hebb's simple "increase in efficiency", is at least approximately correct. The particular learning law presented in this section is now referred to as Hebbian learning, even though his ideas were somewhat more abstract. This learning belongs to the *coincidence* category of learning laws which cause weight changes in response to localized events in neural processing. We describe the learning laws of this category by their local time and space properties.

Hebbian learning has been used in a number of network architectures. It is used in both supervised and unsupervised learning modes. The effect of strengthening the connection between two neurons, when one contributes to the firing of another, may be simulated mathematically by adjusting the weight on their connection by a constant times the sign of the product of their output values.

Let's see how this works. Suppose neurons i and j are connected so that the output of i is an input of j. We can define the weight adjustment on the connection between them, ΔW, as the sign of $c * (o_i * o_j)$, where c is a constant controlling the learning rate. In Table 11.4,

O_i is the sign of the output value of i and O_j of the output of j. From the first line of the table we see that when O_i and O_j are both positive, the weight adjustment, ΔW, is positive. This has the effect of strengthening the connection between i and j when i has contributed to j's "firing."

O_i	O_j	$O_i * O_j$
+	+	+
+	−	−
−	+	−
−	−	+

Table 11.4 The signs and product of signs of node output values.

In the second and third rows of Table 11.4, i and j have opposite signs. Since their signs differ, we want to inhibit i's contribution to j's output value. Therefore we adjust the weight of the connection by a negative increment. Finally, in the fourth row, i and j again have the same sign. This means that we increase the strength of their connection. This weight adjustment mechanism has the effect of reinforcing the path between neurons when they have similar signals and inhibiting them otherwise.

In the next sections we consider two types of Hebbian learning, unsupervised and supervised. We begin by examining an unsupervised form.

11.5.2 An Example of Unsupervised Hebbian Learning

Recall that in unsupervised learning a critic is not available to provide the "correct" output value; thus the weights are modified solely as a function of the input and output values of the neuron. The training of this network has the effect of strengthening the network's responses to patterns that it has already seen. In the next example, we show how Hebbian techniques can be used to model conditioned response learning, where an arbitrarily selected stimulus can be used as a condition for a desired response.

Weight can be adjusted, ΔW, for a node i in unsupervised Hebbian learning with:

$$\Delta W = c * f(X, W) * X$$

where c is the learning constant, a small positive number, f(X, W) is i's output, and X is the input vector to i.

We now show how a network can use Hebbian learning to transfer its response from a primary or unconditioned stimulus to a conditioned stimulus. This allows us to model the type of learning studied in Pavlov's experiments, where by simultaneously ringing a bell every time food was presented, a dog's salivation response to food was transferred to the bell. The network of Figure 11.19 has two layers, an input layer with six nodes and an output layer with one node.The output layer returns either +1, signifying that the output neuron has fired, or a −1, signifying that it is quiescent.

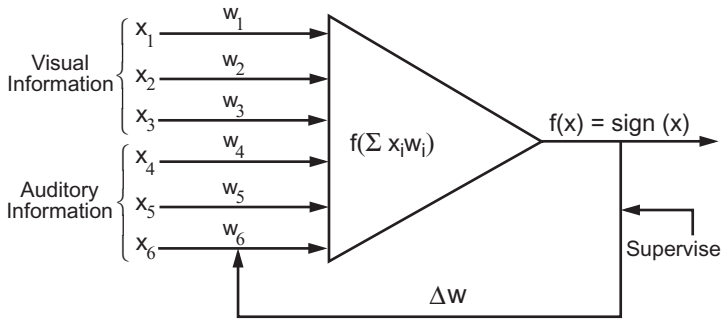


Figure 11.19 An example neuron for application of a hybrid Hebbian node where learning is supervised.

We let the learning constant be the small positive real number 0.2. In this example we train the network on the pattern [1, –1, 1, –1, 1 –1] which is the concatenation of the two patterns, [1, –1, 1] and [–1, 1, –1]. The pattern [1, –1, 1] represents the unconditioned stimulus and [–1, 1, –1] represents the new stimulus.

We assume that the network already responds positively to the unconditioned stimulus but is neutral with respect to the new stimulus. We simulate the positive response of the network to the unconditioned stimulus with the weight vector [1, –1, 1], exactly matching the input pattern, while the neutral response of the network to the new stimulus is simulated by the weight vector [0, 0, 0]. The concatenation of these two weight vectors gives us the initial weight vector for the network, [1, –1, 1, 0, 0, 0].

We now train the network on the input pattern, hoping to induce a configuration of weights which will produce a positive network response to the new stimulus. The first iteration of the network gives:

$$W * X = (1 * 1) + (-1 * -1) + (1 * 1) + (0 * -1) + (0 * 1) + (0 * -1)$$
$$= (1) + (1) + (1) = 3$$
$$f(3) = \text{sign}(3) = 1.$$

We now create the new weight W^2:

$$W^2 = [1, -1, 1, 0, 0, 0] + .2 * (1) * [1, -1, 1, -1, 1, -1]$$
$$= [1, -1, 1, 0, 0, 0] + [.2, -.2, .2, -.2, .2, -.2]$$
$$= [1.2, -1.2, 1.2, -.2, .2, -.2.]$$

We expose the adjusted network to the original input pattern:

$$W * X = (1.2 * 1) + (-1.2 * -1) + (1.2 * 1) + (-.2 * -1) + (.2 * 1) + (-.2 * -1)$$
$$= (1.2) + (1.2) + (1.2) + (+.2) + (.2) + (.2) = 4.2 \text{ and}$$
$$\text{sign}(4.2) = 1.$$

We now create the new weight W^3:

W^3 = [1.2, –1.2, 1.2, –.2, .2, –.2] + .2 * (1) * [1, –1, 1, –1, 1 –1]
= [1.2, –1.2, 1.2, –.2, .2, –.2] + [.2, –.2, .2, –.2, .2, –.2]
= [1.4, –1.4, 1.4, –.4, .4, –.4.]

It can now be seen that the vector product, W * X, will continue to grow in the positive direction, with the absolute value of each element of the weight vector increasing by .2 at each training cycle. After 10 more iterations of Hebbian training the weight vector will be:

W^{13} = [3.4, –3.4, 3.4, –2.4, 2.4, –2.4].

We now use this trained weight vector to test the network's response to the two partial patterns. We would like to see if the network continues to respond to the unconditioned stimulus positively and, more importantly, if the network has now acquired a positive response to the new, conditioned stimulus. We test the network first on the unconditioned stimulus [1, –1, 1]. We fill out the last three arguments of the input vector with random 1, and –1 assignments. For example, we test the network on the vector [1, –1, 1, 1, 1, –1]:

sign(W*X) = sign((3.4*1) + (–3.4*–1) + (3.4*1)
+ (–2.4*1) + (2.4*1) + (–2.4*–1))
= sign(3.4 + 3.4 + 3.4 – 2.4 + 2.4 + 2.4)
= sign(12.6) = +1.

The network thus still responds positively to the original unconditioned stimulus. We now do a second test using the original unconditioned stimulus and a different random vector in the last three positions [1, –1, 1, 1, –1, –1]:

sign(W*X) = sign((3.4*1) + (–3.4*–1) + (3.4*1)
+ (–2.4*1) + (2.4*–1) + (–2.4*–1))
= sign(3.4 + 3.4 + 3.4 – 2.4 – 2.4 + 2.4)
= sign(7.8) = +1.

The second vector also produces a positive network response. In fact we note in these two examples that the network's sensitivity to the original stimulus, as measured by its raw activation, has been strengthened, due to repeated exposure to that stimulus.

We now test the network's response to the new stimulus pattern, [–1, 1, –1], encoded in the last three positions of the input vector. We fill the first three vector positions with random assignments from the set {1,–1} and test the network on the vector [1, 1, 1, –1, 1, –1]:

sign(W*X) = sign((3.4*1) + (–3.4*–1) + (3.4*1)
+ (–2.4*1) + (2.4*1) + (–2.4*–1))
= sign(3.4 – 3.4 + 3.4 + 2.4 + 2.4 + 2.4)
= sign(10.6) = +1.

The pattern of the secondary stimulus is also recognized!

We do one final experiment, with the vector patterns slightly degraded. This could represent the stimulus situation where the input signals are slightly altered, perhaps because a new food and a different sounding bell are used. We test the network on the input vector [1, –1, –1, 1, 1, –1], where the first three parameters are one off the original unconditioned stimulus and the last three parameters are one off the conditioned stimulus:

```
sign(W*X)  = sign((3.4*1) + (–3.4*–1) + (3.4*1)
               + (–2.4*1) + (2.4*1) + (–2.4*–1))
           = sign(3.4 + 3.4 – 3.4 – 2.4 + 2.4 + 2.4)
           = sign(5.8) = +1.
```

Even the partially degraded stimulus is recognized!

What has the Hebbian learning model produced? We created an association between a new stimulus and an old response by repeatedly presenting the old and new stimulus together. The network learns to transfer its response to the new stimulus without any supervision. This strengthened sensitivity also allows the network to respond in the same way to a slightly degraded version of the stimuli. This was achieved by using Hebbian coincidence learning to increase the strength of the network's response to the total pattern, an increase which has the effect of increasing the strength of the network's response to each individual component of the pattern.

11.5.3 Supervised Hebbian Learning

The Hebbian learning rule is based on the principle that the strength of the connection between neurons is increased whenever one neuron contributes to the firing of another. This principle can be adapted to a supervised learning situation by basing the connection weight adjustment on the desired output of the neuron rather than the actual output. For example, if the input of neuron A to neuron B is positive, and the *desired* response of neuron B is a positive output, then the weight on the connection from A to B is increased.

We examine an application of supervised Hebbian learning showing how a network can be trained to recognize a set of associations between patterns. The associations are given by a set of ordered pairs, $\{<X_1, Y_1>, <X_2, Y_2>, ..., <X_t, Y_t>\}$, where X_i and Y_i are the vector patterns to be associated. Suppose that the length of the X_i is n and the Y_i is m. We design the network to explicitly fit this situation. Therefore, it has two layers, an input layer of size n and an output layer of size m, as in Figure 11.20.

The learning formula for this network can be derived by starting with the Hebbian learning formula from the previous section:

$$\Delta W = c * f(X, W) * X$$

where f(X, W) is the actual output of the network node. In supervised learning, we replace this actual output of a node with the desired output vector D, giving us the formula:

$$\Delta W = c * D * X$$

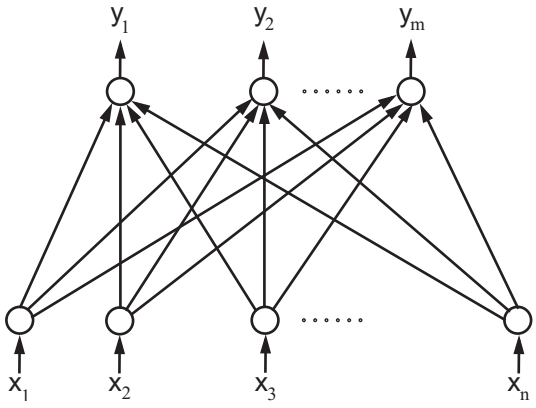


Figure 11.20 A supervised Hebbian network for learning pattern association.

Given a vector pair, <X,Y> from the set of associated pairs, we apply this learning rule to the kth node in the output layer:

$$\Delta W_{ik} = c * d_k * x_i,$$

where ΔW_{ik} is the weight adjustment on the ith input to the kth node in the output layer, d_k is the desired output of the kth node, and x_i is the ith element of X. We apply this formula to adjust all the weights on all the nodes in the output layer. The vector $<x_1, x_2, .., x_n>$ is just the input vector X and the vector $<d_1, d_2, .., d_m>$ is the output vector Y. Applying the formula for individual weight adjustments across the entire output layer and collecting terms, we can write the formula for the weight adjustment on the output layer as:

$$\Delta W = c * Y * X,$$

where the vector product Y*X is the *outer vector product*. The outer vector product YX is defined in general as the matrix:

$$YX = \begin{bmatrix} y_1 \bullet x_1 & y_1 \bullet x_2 & \cdots & y_1 \bullet x_m \\ y_2 \bullet x_1 & y_2 \bullet x_2 & \cdots & y_2 \bullet x_m \\ \cdots & \cdots & \cdots & \cdots \\ y_n \bullet x_1 & y_n \bullet x_2 & \cdots & y_n \bullet x_m \end{bmatrix}$$

To train the network on the entire set of associated pairs, we cycle through these pairs, adjusting the weight for each pair $<X_i, Y_i>$ according to the formula:

$$W^{t+1} = W^t + c*Y_i*X_i.$$

For the entire training set we get:

$$W^1 = W^0 + c\,(Y_1 * X_1 + Y_2 * X_2 + \ldots + Y_t * X_t),$$

where W^0 is the initial weight configuration. If we then initialize W^0 to the 0 vector, <0, 0, ...,0>, and set the learning constant c to 1, we get the following formula for assigning network weights:

$$W = Y_1 * X_1 + Y_2 * X_2 + \ldots + Y_t * X_t.$$

A network which maps input vectors to output vectors using this formula for weight assignment is called a *linear associator*. We have shown that linear associator networks are based on the Hebbian learning rule. In practice this formula can be applied directly to initialize network weights without explicit training.

We next analyze the properties of the linear associator. This model, as we have just seen, stores multiple associations in a matrix of weight vectors. This raises the possibility of interactions between stored patterns. We analyze the problems created by these interactions in the next sections.

11.5.4 Associative Memory and the Linear Associator

The *linear associator* network was first proposed by Tuevo Kohonen (1972) and James Anderson et al. (1977). In this section we present the linear associator network as a method for storing and recovering patterns from memory. We examine different forms of memory retrieval, including the heteroassociative, autoassociative, and the interpolative models. We analyze the linear associator network as an implementation of interpolative memory based on Hebbian learning. We end this section by considering problems with interference or crosstalk. This problem can arise when encoding multiple patterns in memory.

We begin our examination of memory with some definitions. Patterns and memory values are represented as vectors. There is always an inductive bias in reducing the representation of a problem to a set of feature vectors. The associations which are to be stored in memory are represented as sets of vector pairs, $\{<X_1, Y_1>, <X_2, Y_2>, \ldots, <X_t, Y_t>\}$. For each vector pair $<X_i, Y_i>$, the X_i pattern is a key for retrieval of the Y_i pattern. There are three types of associative memories:

1. *Heteroassociative*: This is a mapping from X to Y such that if an arbitrary vector X is closer to the vector X_i than any other exemplar, then the associated vector Y_i is returned.

2. *Autoassociative*: This mapping is the same as the heteroassociative except that $X_i = Y_i$ for all exemplar pairs. Since every pattern X_i is related to itself, this form of memory is primarily used when a partial or degraded stimulus pattern serves to recall the full pattern.

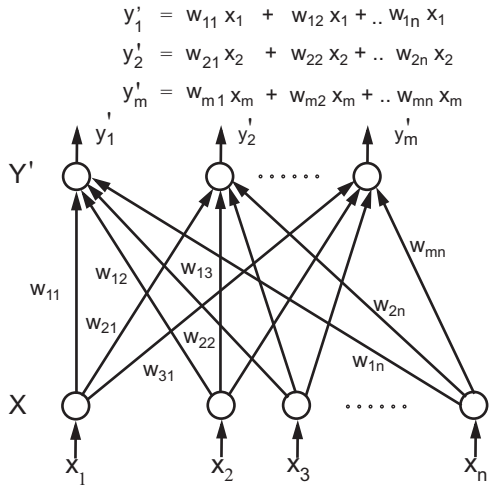


Figure 11.21 The linear association network. The vector X_i is entered as input and the associated vector Y' is produced as output. y'_i is a linear combination of the x input. In training each y'_i is supplied with its correct output signals.

3. *Interpolative*: This is a mapping Φ of X to Y such that when X differs from an exemplar, that is, $X = X_i + \Delta_i$, then the output of the $\Phi(X) = \Phi(X_i + \Delta_i) = Y_i + E$, where $E = \Phi(\Delta_i)$. If the input vector is one of the exemplars X_i the associated Y_i is retrieved. If it differs from an exemplar by the vector Δ, then the output vector also differs by the vector difference E, where $E = \Phi(\Delta)$.

The autoassociative and heteroassociative memories are used for retrieval of one of the original exemplars. They constitute memory in the true sense, in that the pattern that is retrieved is a literal copy of the stored pattern. We also may want to construct an output pattern that differs from the patterns stored in memory in some systematic way. This is the function of an interpolative memory.

The linear associator network in Figure 11.21 implements a form of interpolative memory. As shown in Section 11.5.3, it is based on the Hebbian learning model. The network weight initialization is described by the equation derived in Section 11.5.3:

$$W = Y_1 * X_1 + Y_2 * X_2 + \ldots + Y_t * X_t.$$

Given this weight assignment, the network will retrieve with an exact match one of the exemplars; otherwise it produces an interpolative mapping.

We next introduce some concepts and notation to help us analyze the behavior of this network. First we want to introduce a metric that allows us to define precisely distance between vectors. All our pattern vectors in the examples are *Hamming* vectors, that is vectors composed of +1 and −1 values only. We use *Hamming distance* to describe the distance between two Hamming vectors. Formally, we define a Hamming space:

$H^n = \{X = (x_1, x_2, ..., x_n)\}$, where each x_i is from the set $\{+1, -1\}$.

Hamming distance is defined for any two vectors from a Hamming space as:

$||X, Y||$ = the number of components by which X and Y differ.

For example, the Hamming distance, in four-dimensional Hamming space, between:

(1, –1, –1, 1) and (1, 1, –1, 1) is 1
(–1, –1, –1, 1) and (1, 1, 1, –1) is 4
(1, –1, 1, –1) and (1, –1, 1, –1) is 0.

We need two further definitions. First, the complement of a Hamming vector is that vector with each of its elements changed: +1 to –1 and –1 to +1. For example, the complement of (1, –1, –1, –1) is (–1, 1, 1, 1).

Second, we define the *orthonormality* of vectors. Vectors that are orthonormal are orthogonal, or perpendicular, and of unit length. Two orthonormal vectors, when multiplied together with the *dot product*, have all their cross-product terms go to zero. Thus, in an orthonormal set of vectors, when any two vectors, X_i and X_j, are multiplied the product is 0, unless they are the same vector:

$X_i X_j = \delta_{ij}$ where $\delta_{ij} = 1$ when $i = j$ and 0 otherwise.

We next demonstrate that the linear associator network defined above has the following two properties, with $\Phi(X)$ representing the mapping function of the network. First, for an input pattern X_i which exactly matches one of the exemplars, the network output, $\Phi(X_i)$, is Y_i, the associated exemplar. Second, for an input pattern X_k, which does not exactly match one of the exemplars, the network output, $\Phi(X_k)$, is Y_k, that is the linear interpolation of X_k. More precisely, if $X_k = X_i + \Delta_i$, where X_i is an exemplar, the network returns:

$Y_k = Y_i + E$, where $E = \Phi(\Delta_i)$.

We first show that, when the network input X_i is one of the exemplars, the network returns the associated exemplar.

$\Phi(X_i) = WX_i$, by the definition of the network activation function.

Since $W = Y_1X_1 + Y_2X_2 + ... + Y_iX_i + ... + Y_nX_n$, we get:

$$\begin{aligned}\Phi(X_i) &= (Y_1X_1 + Y_2X_2 + ... + Y_iX_i + ... + Y_nX_n)X_i \\ &= Y_1X_1X_i + Y_2X_2X_i + ... + Y_iX_iX_i + ... + Y_nX_nX_i, \text{ by distributivity.}\end{aligned}$$

Since, as defined above, $X_iX_j = \delta_{ij}$:

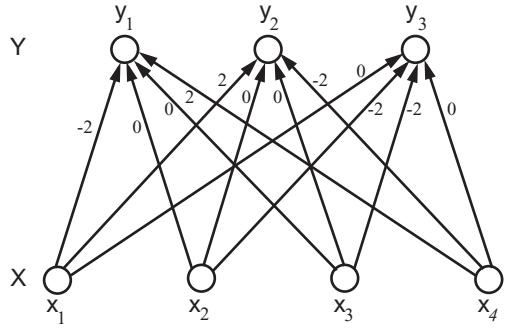


Figure 11.22 A linear associator network for the example in Section 11.5.4. The weight matrix is calculated using the formula presented in the previous section.

$$\Phi(X_i) = Y_1\delta_{1i} + Y_2\delta_{2i} + \ldots + Y_i\delta_{ii} + \ldots + Y_n\delta_{ni}.$$

By the orthonormality condition, $\delta_{ij} = 1$ when $i = j$ and 0 otherwise. Thus we get:

$$\Phi(X_i) = Y_1*0 + Y_2*0 + \ldots + Y_i*1 + \ldots + Y_n*0 = Y_i.$$

It can also be shown that, for X_k not equal to any of the exemplars, the network performs an interpolative mapping. That is, for $X_k = X_i + \Delta_i$, where X_i is an exemplar,

$$\begin{aligned}\Phi(X_k) &= \Phi(X_i + \Delta_i)\\ &= Y_i + E,\end{aligned}$$

where Y_i is the vector associated with X_i and

$$E = \Phi(\Delta_i) = (Y_1X_1 + Y_2X_2 + \ldots + Y_nX_n)\,\Delta_i.$$

We omit the details of the proof.

We now give an example of linear associator processing. Figure 11.22 presents a simple linear associator network that maps a four-element vector X into a three-element vector Y. Since we are working in a Hamming space, the network activation function f is the *sign* function used earlier.

If we want to store the following two vector associations $\langle X_1, Y_1\rangle$, $\langle X_2, Y_2\rangle$ and:

$$X_1 = [1, -1, -1, -1] \leftrightarrow Y_1 = [-1, 1, 1],$$
$$X_2 = [-1, -1, -1, 1] \leftrightarrow Y_2 = [1, -1, 1].$$

Using the weight initialization formula for linear associators, with the outer vector product as defined in the previous section:

$$W = Y_1X_1 + Y_2X_2 + Y_3X_3 + \ldots + Y_nX_n,$$

We can now calculate $Y_1X_1 + Y_2X_2$, the weight matrix for the network:

$$W = \begin{bmatrix} -1 & 1 & 1 & 1 \\ 1 & -1 & -1 & -1 \\ 1 & -1 & -1 & -1 \end{bmatrix} + \begin{bmatrix} -1 & -1 & -1 & 1 \\ 1 & 1 & 1 & -1 \\ -1 & -1 & -1 & 1 \end{bmatrix} = \begin{bmatrix} -2 & 0 & 0 & 2 \\ 2 & 0 & 0 & -2 \\ 0 & -2 & -2 & 0 \end{bmatrix}$$

We run the linear associator on one of the exemplars. We start with $X = [1, -1, -1, -1]$ from the first exemplar pair to get back the associated Y:

$y_1 = (-2*1) + (0*-1) + (0*-1) + (2*-1) = -4$, and $\text{sign}(-4) = -1$,
$y_2 = (2*1) + (0*-1) + (0*-1) + (-2*-1) = 4$, and $\text{sign}(4) = 1$, and
$y_3 = (0*1) + (-2*-1) + (-2*-1) + (0*-1) = 4$, and $\text{sign}(4) = 1$.

Thus $Y_1 = [-1, 1, 1]$, the other half of the exemplar pair, is returned.

We next show an example of linear interpolation of an exemplar. Consider the X vector $[1, -1, 1, -1]$:

$y_1 = (-2*1) + (0*-1) + (0*1) + (2*-1) = -4$, and $\text{sign}(-4) = -1$,
$y_2 = (2*1) + (0*-1) + (0*1) + (-2*-1) = 4$, and $\text{sign}(4) = 1$, and
$y_3 = (0*1) + (-2*-1) + (-2*1) + (0*-1) = 0$, and $\text{sign}(0) = 1$.

Notice that $Y = [-1, 1, 1]$ is not one of the original Y exemplars. Notice that the mapping preserves the values which the two Y exemplars have in common. In fact $[1, -1, 1, -1]$, the X vector, has a Hamming distance of 1 from each of the two X exemplars; the output vector $[-1, 1, 1]$ also has a Hamming distance of 1 from each of the other Y exemplars.

We summarize with a few observations regarding linear associators. The desirable properties of the linear associator depend on the requirement that the exemplar patterns comprise an orthonormal set. This restricts its practicality in two ways. First, there may be no obvious mapping from situations in the world to orthonormal vector patterns. Second, the number of patterns which can be stored is limited by the dimensionality of the vector space. When the orthonormality requirement is violated, interference between stored patterns occurs, causing a phenomenon called *crosstalk*.

Observe also that the linear associator retrieves an associated Y exemplar only when the input vector exactly matches an X exemplar. When there is not an exact match on the input pattern, the result is an interpolative mapping. It can be argued that interpolation is not memory in the true sense. We often want to implement a true memory retrieval function where an approximation to an exemplar retrieves the exact pattern that is associated with it. What is required is a *basin* of attraction to capture vectors in the surrounding region.

In the next section, we demonstrate an *attractor* version of the linear associator network.

11.6 Attractor Networks or "Memories"

11.6.1 Introduction

The networks discussed to this point are *feedforward*. In feedforward networks information is presented to a set of input nodes and the signal moves forward through the nodes or layers of nodes until some result emerges. Another important class of connectionist networks are *feedback* networks. The architecture of these nets is different in that the output signal of a node can be cycled back, directly or indirectly, as input to that node.

Feedback networks differ from feedforward networks in several important ways:

1. the presence of feedback connections between nodes,
2. a time delay, i.e., noninstantaneous signal propagation,
3. output of the network is the network's state upon convergence,
4. network usefulness depends on convergence properties.

When a feedback network reaches a time in which it no longer changes, it is said to be in a state of equilibrium. The state which a network reaches on equilibrium is considered to be the network output.

In the feedback networks of Section 11.6.2, the network state is initialized with an input pattern. The network processes this pattern, passing through a series of states until it reaches equilibrium. The network state on equilibrium is the pattern retrieved from memory. In Section 11.6.3, we consider networks that implement a heteroassociative memory, and in Section 11.6.4, an autoassociative memory.

The cognitive aspects of these memories are both interesting and important. They offer us a model for content addressable memory. This type of associator can describe the retrieval of a phone number, the feeling of sadness from an old memory, or even the recognition of a person from a partial facial view. Researchers have attempted to capture many of the associative aspects of this type of memory in symbol-based data structures, including semantic networks, frames, and object systems, as seen in Chapter 6.

An *attractor* is defined as a state toward which states in a neighboring region evolve across time. Each attractor in a network will have a region where any network state inside that region evolves toward that attractor. That region is called its *basin*. An attractor can consist in a single network state or a series of states through which the network cycles.

Attempts to understand attractors and their basins mathematically have given rise to the notion of a network energy function (Hopfield 1984). Feedback networks with an energy function that has the property that every network transition reduces total network energy are guaranteed to converge. We describe these networks in Section 11.6.3.

Attractor networks can be used to implement content addressable memories by installing the desired patterns as attractors in memory. They can also be used to solve optimization problems, such as the traveling salesperson problem, by creating a mapping between the cost function in the optimization problem and the network energy. The solution of the problem then comes through the reduction of total network energy. This type of problem solving is done with what is called a Hopfield network.

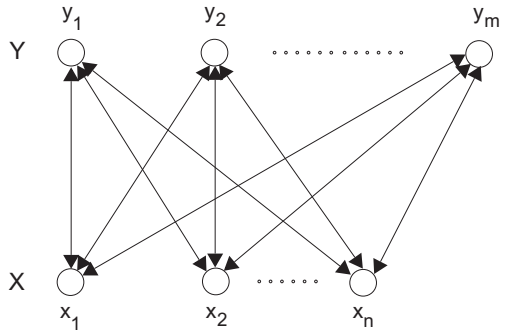


Figure 11.23 A BAM network for the examples of Section 11.6.2. Each node may also be connected to itself.

11.6.2 BAM, the Bi-directional Associative Memory

The BAM network, first described by Bart Kosko (1988), consists of two fully interconnected layers of processing elements. There can also be a feedback link connecting each node to itself. The BAM mapping of an n dimensional input vector X_n into the m dimensional output vector Y_m is presented in Figure 11.22. Since each link from X to Y is bi-directional, there will be weights associated with the information flow going in each direction.

Like the weights of the linear associator, the weights on the BAM network can be worked out in advance. In fact we use the same method for calculating network weights as that used in the linear associator. The vectors for the BAM architecture are taken from the set of Hamming vectors.

Given the N vector pairs that make up the set of exemplars we wish to store, we build the matrix as we did in Section 11.5.4:

$$W = Y_1 * X_1 + Y_2 * X_2 + ... + Y_t * X_t.$$

This equation gives the weights on the connections from the X layer to the Y layer, as can be seen in Figure 11.23. For example, w_{32} is the weight on the connection from the second unit on the X layer to the third unit on the Y layer. We assume that any two nodes only have one pathway between them. Therefore, the weights connecting nodes on the X and Y layers are identical in both directions. Thus, the weight matrix from Y to X is the transpose of the weight matrix W.

The BAM network can be transformed into an autoassociative network by using the same weight initialization formula on the set of associations $<X_1, X_1>$, $<X_2, X_2>$, .. . Since the X and Y layers resulting from this procedure are identical we can eliminate the Y layer, resulting in a network which looks like Figure 11.24. We look at an example of an autoassociative network in Section 11.6.4.

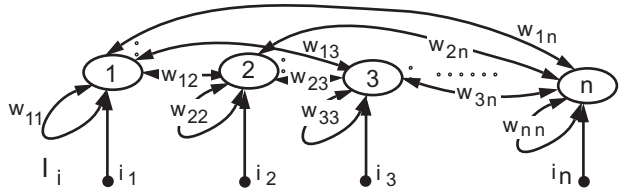


Figure 11.24 An autoassociative network with an input vector I_i. We assume single links between nodes with unique indices, thus $w_{ij} = w_{ji}$ and the weight matrix is symmetric.

The BAM network is used to retrieve patterns from memory by initializing the X layer with an input pattern. If the input pattern is a noisy or incomplete version of one of the exemplars, the BAM can often complete the pattern and retrieve the associated pattern.

To recall data with BAM, we do the following:

1. Apply an initial vector pair (X, Y) to the processing elements. X is the pattern for which we wish to retrieve an exemplar. Y is randomly initialized.

2. Propagate the information from the X layer to the Y layer and update the values at the Y layer.

3. Send the updated Y information back to the X layer, updating the X units.

4. Continue the preceding two steps until the vectors stabilize, that is until there is no further changes in the X and Y vector values.

The algorithm just presented gives BAM its feedback flow, its bidirectional movement toward equilibrium. The preceding set of instructions could have begun with a pattern at the Y level leading, upon convergence, to the selection of an X vector exemplar. It is fully bidirectional: we can take an X vector as input and can get a Y association on convergence or we can take a Y vector as input and get back a X association. We will see these issues worked through with an example in the next section.

Upon convergence, the final equilibrium state gives back one of the exemplars used to build the original weight matrix. If all goes as expected, we take a vector of known properties, either identical to or slightly different, from one of the exemplar vector pairs. We use this vector to retrieve the other vector in the exemplar pair. The distance is Hamming distance measured by component-wise comparison of the vectors, counting one for each element difference. Because of the orthonormality constraints, when BAM converges for a vector, it also converges for its complement. Thus we note that the complement of the vector also becomes an attractor. We give an example of this in the next section.

There are several things that can interfere with the BAM convergence. If too many exemplars are mapped into the weight matrix, the exemplars themselves can be too close together and produce pseudo-stabilities in the network. This phenomenon is called *crosstalk,* and occurs as local minima in the network energy space.

We next consider briefly the BAM processing. The multiplication of an input vector by the weight matrix computes the sums of the pairwise vector products of the vectors for

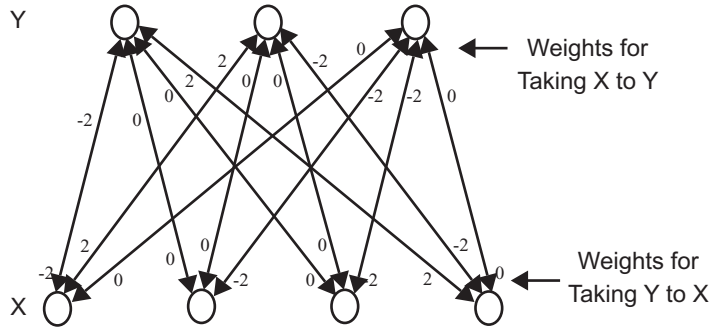


Figure 11.25 A BAM network for the examples of Section 11.6.3.

each element of the output vector. A simple thresholding function then translates the resultant vector back to a vector in the Hamming space. Thus:

net (Y) = WX, or for each Y_i component, net $(Y_i) = \Sigma w_{ij} * x_j$,

with similar relationships for the X layer. The thresholding function f for net(Y) at the time t +1 is also straightforward:

$$f(net^{t+1}) = \begin{cases} +1 & \text{if net} > 0 \\ f(net^{t}) & \text{if net} = 0 \\ -1 & \text{if net} < 0 \end{cases}$$

In the next section we illustrate this *bidirectional associative memory* processing with several examples.

11.6.3 Examples of BAM Processing

Figure 11.25 presents a small BAM network, a simple variation of the linear associator presented in Section 11.5.4. This network maps a four element vector X into a three element vector Y and vice versa. Suppose we want to create the two vector pair exemplars:

$x_1 = [1, -1, -1, -1] \leftrightarrow y_1 = [1, 1, 1]$, and
$x_2 = [-1, -1, -1, 1] \leftrightarrow y_2 = [1, -1, 1]$.

We now create the weight matrix using the formula presented in the previous section:

$$W = Y_1X_1^t + Y_2X_2^t + Y_3X_3^t + \ldots + Y_NX_N^t$$

$$W = \begin{bmatrix} 1 & -1 & -1 & -1 \\ 1 & -1 & -1 & -1 \\ 1 & -1 & -1 & -1 \end{bmatrix} + \begin{bmatrix} -1 & -1 & -1 & 1 \\ 1 & 1 & 1 & -1 \\ -1 & -1 & -1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -2 & -2 & 0 \\ 2 & 0 & 0 & -2 \\ 0 & -2 & -2 & 0 \end{bmatrix}$$

The weight vector for the mapping from Y to X is the transpose of W, or:

$$\begin{bmatrix} 0 & 2 & 0 \\ -2 & 0 & -2 \\ -2 & 0 & -2 \\ 0 & -2 & 0 \end{bmatrix}$$

We now select several vectors and test the BAM associator. Let's start with an exemplar pair, choosing the X component and seeing if we get the Y. Let X = [1, –1, –1, –1]:

Y_1 = (1*0) + (–1*–2) + (–1*–2) + (0*–1) = 4, and f(4) = 1,
Y_2 = (1*2) + (–1*0) + (–1*0) + (–1*–2) = 4, and f(4) = 1, and
Y_3 = (1*0) + (–1*–2) + (–1*–2) + (–1*0) = 4, and f(4) = 1.

Thus the other half of the exemplar pair is returned. The reader can make this Y vector an input vector and verify that the original X vector [1, –1, –1, –1] is returned.

For our next example, consider the X vector [1, 1, 1, –1], with Y randomly initialized. We map X with our BAM network:

Y_1 = (1*0) + (1*–2) + (1*–2) + (–1*0) = –4, and f(4) = –1,
Y_2 = (1*2) + (1*0) + (1*0) + (–1*–2) = 4, and f(4) = 1,
Y_3 = (1*0) + (1*–2) + (1*–2) + (–1*0) = –4, and f(4) = –1.

This result, with the thresholding function f applied to [–4, 4, –4], is [–1, 1, –1]. Mapping back to X gives:

X_1 = (–1*0) + (1*2) + (–1* 0) = 2,
X_2 = (–1*–2) + (1*0) + (–1*–2) = 4,
X_3 = (–1*–2) + (1*0) + (–1*–2) = 4,
X_4 = (–1*0) + (1*–2) + (–1*0) = –2.

The threshold function applied, again as above, gives the original vector [1, 1, 1, –1]. Since the starting vector produced a stable result with its first translation, we might think we have just discovered another prototype exemplar pair. In fact, the example we selected is the complement of the original <X_2, Y_2> vector exemplar! It turns out that in a BAM network, when a vector pair is established as an exemplar prototype, so is its complement. Therefore, our BAM network includes two more prototypes:

X_3 = [–1, 1, 1, 1] ↔ Y_3 = [–1, –1, –1], and
X_4 = [1, 1, 1, –1] ↔ Y_4 = [–1, 1, –1].

Let us next select a vector near an X exemplar, [1, –1, 1, –1]. Note that the Hamming distance from the closest of the four X exemplars is 1. We next randomly initialize the vector Y to [–1, –1, –1]:

$Y_1^{t+1} = (1*0) + (-1*—2) + (1*2) + (-1*0) = 0,$
$Y_2^{t+1} = (1*2) + (-1*0) + (1*0) + (-1*-2) = 4,$
$Y_3^{t+1} = (1*0) + (-1*-2) + (1*-2) + (-1*0) = 0.$

The evaluation of the net function $f(Y_i^{t+1}) = f(Y_i^t)$ when $y_i^{t+1} = 0$, from the threshold equation at the end of Section 11.6.2. Thus, Y is [−1, 1, −1] due to the random initialization of the first and third parameters of the Y^T to −1. We now take Y back to X:

$X_1 = (-1*0) + (1*2) + (-1* 0) = 2,$
$X_2 = (-1*-2) + (1*0) + (-1*-2) = 4,$
$X_3 = (-1*-2) + (1*0) + (-1*-2) = 4,$
$X_4 = (-1*0) + (1*-2) + (-1*0) = -2.$

The threshold function maps this result to the vector X = [1, 1, 1, −1]. We repeat the process taking this vector back to Y:

$Y_1 = (1*0) + (1*-2) + (1*-2) + (-1*0) = -4,$
$Y_2 = (1*2) + (1*0) + (1*0) + (-1*-2) = 4,$
$Y_3 = (1*0) + (1*-2) + (1*-2) + (-1*0) = -4.$

The threshold function applied to [−4, 4, −4] again gives Y = [−1, 1, −1]. This vector is identical to the most recent version of Y, so the network is stable. This demonstrates that after two passes through the BAM net, a pattern that was close to X_4 converged to the stored exemplar. This would be similar to recognizing a face or other stored image with part of the information missing or obscured. The Hamming distance between the original X vector [1, −1, 1, −1] and the X_4 prototype [1, 1, 1, −1] was 1. The vector settled into the $<X_4, Y_4>$ exemplar pair.

In our BAM examples we started processing with the X element of the exemplar pair. Of course, we could have designed the examples from the Y vector, initializing X when necessary.

Hecht-Nielsen (1990, p. 82) presents an interesting analysis of the BAM network. He demonstrates that the orthonormal property for the linear associator network support for BAM is too restrictive. He gives an argument showing that the requirement for building the network is that the vectors be linearly independent, that is, that no vector can be created from a linear combination of other vectors in the space of exemplars.

11.6.4 Autoassociative Memory and Hopfield Nets

The research of John Hopfield, a physicist at California Institute of Technology, is a major reason connectionist architectures have their current credibility. He studied network convergence properties, using the concept of energy minimization. He also designed a family of networks based on these principles. As a physicist, Hopfield understood stabilities of physical phenomena as energy minimization points of the physical system. An example of this approach is the simulated annealing analysis of the cooling of metals.

Let us first review the basic characteristics of feedback associative networks. These networks begin with an initial state consisting of the input vector. The network then processes this signal through feedback pathways until it reaches a stable state. To use this architecture as an associative memory we would like the network to have two properties. First, starting from any initial state we would like a guarantee that the network will converge on some stable state. Second, we would like this stable state to be the one closest to the input state by some distance metric.

We look first at an autoassociative network built on the same principles as the BAM network. We noted in the previous section that BAM networks can be transformed into autoassociative networks by using identical vectors in the X and Y positions. The result of this transformation, as we see next, is a symmetric square weight matrix. Figure 11.23 of Section 11.6.2 offered an example.

The weight matrix for the autoassociative network that stores a set of vector exemplars $\{X_1, X_2, ..., X_n\}$ is created by:

$$W = \Sigma X_i X_i^t \qquad \text{for } i = 1, 2, ..., n.$$

When we create the autoassociative memory from the heteroassociative, the weight from node x_i to x_j will be identical to that from x_j to x_i and so the weight matrix will be symmetric. This assumption only requires that the two processing elements be connected by one path having a single weight. We may also have the special case, again with neural plausibility, that no network node is directly linked to itself, that is, there are no x_i to x_i links. In this situation the main diagonal of the weight matrix, w_{ij} where $i = j$, is all zeros.

As with BAM, we work out the weight matrix based on the patterns to be stored in memory. We clarify this with a simple example. Consider the three vector exemplar set:

$X_1 = [1, -1, 1, -1, 1]$,
$X_2 = [-1, 1, 1, -1, -1]$,
$X_3 = [1, 1, -1, 1, 1]$.

We next calculate the weight matrix using $W = \Sigma X_i X_i^t$ for $i = 1, 2, 3$:

$$W = \begin{bmatrix} 1 & -1 & 1 & -1 & 1 \\ -1 & 1 & -1 & 1 & -1 \\ 1 & -1 & 1 & -1 & 1 \\ -1 & 1 & -1 & 1 & -1 \\ 1 & -1 & 1 & -1 & 1 \end{bmatrix} + \begin{bmatrix} 1 & -1 & -1 & 1 & 1 \\ -1 & 1 & 1 & -1 & -1 \\ -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 1 & -1 & 1 & 1 \\ 1 & 1 & -1 & 1 & 1 \\ -1 & -1 & 1 & -1 & -1 \\ 1 & 1 & -1 & 1 & 1 \\ 1 & 1 & -1 & 1 & 1 \end{bmatrix}$$

$$W = \begin{bmatrix} 3 & -1 & -1 & 1 & 3 \\ -1 & 3 & -1 & 1 & -1 \\ -1 & -1 & 3 & -3 & -1 \\ 1 & 1 & -3 & 3 & 1 \\ 3 & -1 & -1 & 1 & 3 \end{bmatrix}$$

We use the thresholding function:

$$f(net^{t+1}) = \begin{cases} +1 & \text{if net} > 0 \\ f(net^t) & \text{if net} = 0 \\ -1 & \text{if net} < 0 \end{cases}$$

We first test the network with an exemplar, $X_3 = [1, 1, -1, 1, 1]$, and obtain:

$$X_3 * W = [7, 3, -9, 9, 7],$$

and with the threshold function, [1, 1, –1, 1, 1]. We see this vector stabilizes immediately on itself. This illustrates that the exemplars are themselves stable states or attractors.

We next test a vector which is Hamming distance 1 from the exemplar X_3. The network should return that exemplar. This is equivalent to retrieving a memory pattern from partially degraded data. We select $X = [1, 1, 1, 1, 1]$:

$$X * W = [5, 1, -3, 3, 5].$$

Using the threshold function gives the X_3 vector [1, –1, –1, 1, 1].

We next take a third example, this time a vector whose Hamming distance is 2 away from its nearest prototype, let $X = [1, -1, -1, 1, -1]$. It can be checked that this vector is 2 away from X_3, 3 away from X_1, and 4 away from X_2. We begin:

$$X * W = [3, -1, -5, 5, 3], \text{ which with threshold yields } [1, -1, -1, 1, 1].$$

This doesn't seem to resemble anything, nor is it a stability point, since:

$$[1, -1, -1, 1, 1] * W = [9, -3, -7, 7, 9], \text{ which is } [1, -1, -1, 1, 1].$$

The net is now stable, but not with one of the original stored memories! Have we found another energy minimum? On closer inspection we note that this new vector is the complement of the original X_2 exemplar [–1, 1, 1, –1, –1]. Again, as in the case of the heteroassociative BAM network, our autoassociative network creates attractors for the original exemplars as well as for their complements, in this case we will have six attractors in all.

To this point in our presentation, we have looked at autoassociative networks based on a linear associator model of memory. One of John Hopfield's goals was to give a more general theory of autoassociative networks which would apply to any single-layer feedback network meeting a certain set of simple restrictions. For this class of single layer feedback networks Hopfield proved that there would always exist a network energy function guaranteeing convergence.

A further goal of Hopfield was to replace the discrete time updating model used previously with one that more closely resembles the continuous time processing of actual neurons. A common way to simulate continuous time asynchronous updating in Hopfield

networks is to update nodes individually rather than as a layer. This is done using a random selection procedure for picking the next node to be updated, while also applying some method for ensuring that on average all the nodes in the network will be updated equally often.

The structure of a Hopfield network is identical to that of the autoassociative network above: a single layer of nodes completely connected (see Figure 11.23). The activation and thresholding also work as before. For node i,

$$x_i^{new} = \begin{cases} +1 & \text{if } \sum_j w_{ij} x_j^{old} > T_i, \\ x_i^{old} & \text{if } \sum_j w_{ij} x_j^{old} = T_i, \\ -1 & \text{if } \sum_j w_{ij} x_j^{old} < T_i, \end{cases}$$

Given this architecture, only one further restriction is required to characterize a Hopfield net. If w_{ij} is the weight on the connection into node i from node j, we define a Hopfield network as one which respects the weight restrictions:

$w_{ii} = 0$ for all i,
$w_{ij} = w_{ji}$ for all i, j.

The Hopfield network does not typically have a learning method associated with it. Like the BAM, its weights are usually calculated in advance.

The behavior of Hopfield networks is now better understood than any other class of networks except perceptrons. This is because its behavior can be characterized in terms of a concise energy function discovered by Hopfield:

$$H(X) = -\sum_i \sum_j w_{ij} x_i x_j + 2\sum_i T_i x_i$$

We will now show that this energy function has the property that every network transition reduces the total network energy. Given the fact that H has a predetermined minimum and that each time H decreases it decreases by at least a fixed minimum amount, we can infer that from any state the network converges.

We first show that for an arbitrary processing element k which is the most recently updated, k changes state if and only if H decreases. The change in energy ΔH is:

$$\Delta H = H(X^{new}) - H(X^{old}).$$

Expanding this equation using the definition of H, we get:

$$\Delta H = -\sum_i \sum_j w_{ij} x_i^{new} x_j^{new} - 2\sum_i T_i x_i^{new} + \sum_i \sum_j w_{ij} x_i^{old} x_j^{old} + 2\sum_i T_i x_i^{old}$$

Since only x_k has changed, $x_i^{new} = x_i^{old}$ for i not equal to k. This means that the terms of the sum that do not contain x_k cancel each other out. Rearranging and collecting terms:

$$\Delta H = -2x_k^{new}\sum_j w_{kj}x_j^{new} + 2T_k x_k^{new} + 2x_k^{old}\sum_j w_{kj}x_j^{old} - 2T_k x_k^{old}.$$

Using the fact that $w_{ii} = 0$ and $w_{ij} = w_{ji}$ we can finally rewrite this as:

$$\Delta H = 2(x_k^{old} - x_k^{new})\left[\sum_j w_{kj}x_j^{old} - T_k\right].$$

To show that ΔH is negative we consider two cases. First, suppose x_k has changed from -1 to $+1$. Then the term in square brackets must have been positive to make x_k^{new} be $+1$. Since $x_k^{old} - x_k^{new}$ is equal to -2, ΔH must be negative. Suppose that x_k has changed from 1 to -1. By the same line of reasoning, ΔH must again be negative. If x_k has not changed state, $x_k^{old} - x_k^{new} = 0$ and $\Delta H = 0$.

Given this result, from any starting state the network must converge. Furthermore, the state of the network on convergence must be a local energy minimum. If it were not then there would exist a transition that would further reduce the total network energy and the update selection algorithm would eventually choose that node for updating.

We have now shown that Hopfield networks have one of the two properties which we want in a network that implements associative memory. It can be shown, however, that Hopfield networks do not, in general, have the second desired property: they do not always converge on the stable state nearest to the initial state. There is no general known method for fixing this problem.

Hopfield networks can also be applied to the solution of optimization problems, such as the traveling salesperson problem. To do this the designer needs to find a way to map the cost function of the problem to the Hopfield energy function. By moving to an energy minimum the network will then also be minimizing the cost with respect to a given problem state. Although such a mapping has been found for some interesting problems, including the traveling salesperson problem, in general, this mapping from problem states to energy states is very difficult to discover.

In this section we introduced heteroassociative and autoassociative feedback networks. We analyzed the dynamical properties of these networks and presented very simple examples showing evolution of these systems toward their attractors. We showed how the linear associator network could be modified into an attractor network called the BAM. In our discussion of continuous time Hopfield networks, we saw how network behavior could be described in terms of an energy function. The class of Hopfield networks have guaranteed convergence because every network transition can be shown to reduce total network energy.

There still remain some problems with the energy-based approach to connectionist networks. First, the energy state reached need not be a global minimum of the system. Second, Hopfield networks need not converge to the attractor nearest to the input vector. This makes them unsuitable for implementing content addressable memories. Third, in using Hopfield nets for optimization, there is no general method for creating a mapping of constraints into the Hopfield energy function. Finally, there is a limit to the total number

of energy minima that can be stored and retrieved from a network, and even more importantly, this number cannot be set precisely. Empirical testing of these networks shows that the number of attractors is a small fraction of the number of nodes in the net. These and other topics are ongoing issues for research (Hecht-Nielsen 1990, Zurada 1992, Freeman and Skapura 1991).

Biology-based approaches, such as genetic algorithms and cellular automata, attempt to mimic the learning implicit in the evolution of life forms. Processing in these models is also parallel and distributed. In the genetic algorithm model, for example, a population of patterns represents the candidate solutions to a problem. As the algorithm cycles, this population of patterns "evolves" through operations which mimic reproduction, mutation, and natural selection. We consider these approaches next, in Chapter 12.

11.7 Epilogue and References

We introduced connectionist learning in this chapter. We took an historical perspective in Section 11.1. For further historical perspective see McCulloch and Pitts (1943), Oliver Selfridge (1959), Claude Shannon (1948), and Frank Rosenblatt (1958). Early psychological models are also important, especially those of Donald Hebb (1949). Cognitive scientists continue to explore the relationship between cognition and brain architecture. Sources include *An Introduction to Natural Computation* (Ballard 1997), *Artificial Minds* (Franklin 1995), *The Cognitive Neuroscience of Action* (Jeannerod 1997), and *Rethinking Innateness: A Connectionist Perspective on Development* (Elman et al. 1996).

We have not addressed many important mathematical as well as computational aspects of connectionist architectures. For an overview, we recommend Robert Hecht-Nielsen (1990), James Freeman and David Skapura (1991), Jacek Zurada (1992), and Nello Cristianini and John Shawe-Taylor 2000). An excellent tutorial on Support Vector Machines is presented by Christopher Burges (1988). Neuro-dynammic programming is described by Bertsekas and Tsitsiklis (1996). For connectionist applications consider *Pattern Recognition in Industry* by P. M. Bhagat (2005)

There are many issues, both representational and computational, that the learning research scientist must consider. These include architecture and connectivity selection for the network as well as determining what cognitive parameters of the environment are to be processed and what the results might "mean." There is also the issue of neural–symbol hybrid systems and how these might reflect different aspects of intelligence.

The backpropagation network is probably the most commonly used connectionist architecture, and thus we gave considerable space to its origins, use, and growth. The two volumes of *Parallel Distributed Processing* (Rumelhart et al.1986*b*) give an introduction to neural networks both as computational and cognitive tools. *Neural Networks and Natural Intelligence* (Grossberg 1988) is another thorough treatment of the subject.

There are also further questions for use of the backpropagation networks, including the number of hidden nodes and layers, selecting the training set, fine-tuning the learning constant, the use of bias nodes, and so on. Many of these issues come under the general heading of *inductive bias*: the role of the knowledge, expectations, and tools that the problem solver brings to problem solving. We address many of these issues in Chapter 16.

Many of the original connectionist architecture designers have described their work. These include John Anderson et al. (1977), Stephan Grossberg (1976, 1988), Geoffrey Hinton and Terrance Sejnowski (1986), Robert Hecht-Nielsen (1989, 1990), John Hopfield (1982, 1984), Tuevo Kohonen (1972, 1984), Bart Kosko (1988), and Carver Mead (1989). More recent approaches, including graphical models, are presented by Michael Jordan (1999) and Brendan Frey (1998). A good modern textbook is written by Christopher Bishop (1995).

11.8 Exercises

1. Make a McCulloch–Pitts neuron that can calculate the logic function implies, $\Rightarrow$.
2. Build a perceptron net in LISP and run it on the classification example of Section 11.2.2.
   a. Generate another data set similar to that of Table 11.3 and run your classifier on it.
   b. Take the results of running the classifier and use the weights to determine the specification for the line separating the sets.
3. Build a backpropagation network in LISP or C++ and use it to solve the exclusive-or problem of Section 11.3.3. Solve the exclusive-or problem with a different backpropagation architecture, perhaps having two hidden nodes and no bias nodes. Compare the convergence speeds using the different architectures.
4. Write a Kohonen net in LISP or C++ and use it to classify the data of Table 11.3. Compare your results with those of Sections 11.2.2 and 11.4.2.
5. Write a counterpropagation net to solve the exclusive-or problem. Compare your results with those of backpropagation net of Section 11.3.3. Use your counterpropagation net to discriminate between the classes of Table 11.3.
6. Use a backpropagation net to recognize the ten (hand drawn) digits. One approach would be to build a 4 x 6 array of points. When a digit is drawn on this grid it will cover some elements, giving them value 1, and miss others, value 0. This 24 element vector would be the input value for your net. You would build your own training vectors. Do the same task with a counterpropagation net; compare your results.
7. Select a different input pattern than that we used in Section 11.5.2. Use the unsupervised Hebbian learning algorithm to recognize that pattern.
8. Section 11.5.4, used the linear associator algorithm to make two vector pair associations. Select three (new) vector pair associations and solve the same task. Test whether your linear associator is interpolative; that is, can it associate near misses of the exemplars? Make your linear associator autoassociative.
9. Consider the bidirectional associative memory (BAM) of Section 11.6.3. Change the association pairs given in our example and create the weight matrix for the associations. Select new vectors and test your BAM associator.
10. Describe the differences between the BAM memory and the linear associator. What is *crosstalk* and how can it be prevented?
11. Write a Hopfield net to solve the traveling salesperson problem for ten cities.