the vector of neuronal outputs at time t. The presence of *feedback* in both models is clearly visible in Fig. 14.8.

14.6 MANIPULATION OF ATTRACTORS AS A RECURRENT NETWORK PARADIGM

When the number of neurons, N, is very large, the neurodynamical model described by Eq. (14.16) possesses, except for the effect of noise, the general properties outlined earlier in Section 14.5: very many degrees of freedom, nonlinearity, and dissipation. Accordingly, such a neurodynamical model can have complicated attractor structures and therefore exhibit useful computational capabilities.

The identification of attractors with computational objects (e.g., associative memories, input–output mappers) is one of the foundations of neural network paradigms. In order to implement this idea, we must exercise *control* over the locations of the attractors in the state space of the system. A learning algorithm then takes the form of a nonlinear dynamical equation that manipulates the locations of the attractors for the purpose of encoding information in a desired form, or learning temporal structures of interest. In this way, it is possible to establish an intimate relationship between the physics of the machine and the algorithms of the computation.

One way in which the collective properties of a neural network may be used to implement a computational task is by way of the concept of *energy minimization*. The Hopfield network and the brain-state-in-a-box model, to be considered in Sections 14.7 and 14.10, respectively, are well-known examples of such an approach. Both of these models are energy-minimizing networks; they differ from each other in their areas of application. The Hopfield network is useful as a content addressable memory or an analog computer for solving combinatorial-type optimization problems. The brain-state-in-a-box model, on the other hand, is useful for clustering types of applications. More will be said about these applications in subsequent sections of the chapter.

The Hopfield network and brain-state-in-a-box model are examples of an associative memory with no hidden neurons: An associative memory is an important resource for intelligent behavior. Another neurodynamical model is that of an input–output mapper, the operation of which relies on the availability of hidden neurons. In this latter case, the method of steepest descent is often used to minimize a cost function defined in terms of the network parameters, and thereby to change the attractor locations. This latter application of a neurodynamical model is exemplified by the dynamically driven recurrent networks discussed in the next chapter.

14.7 HOPFIELD MODEL

The *Hopfield network (model)* consists of a set of neurons and a corresponding set of unit delays, forming a *multiple-loop feedback system*, as illustrated in Fig. 14.9. The number of feedback loops is equal to the number of neurons. Basically, the output of each neuron is fed back, via a unit delay element, to each of the other neurons in the network. In other words, there is *no* self-feedback in the network; the reason for avoiding the use of self-feedback is explained later.

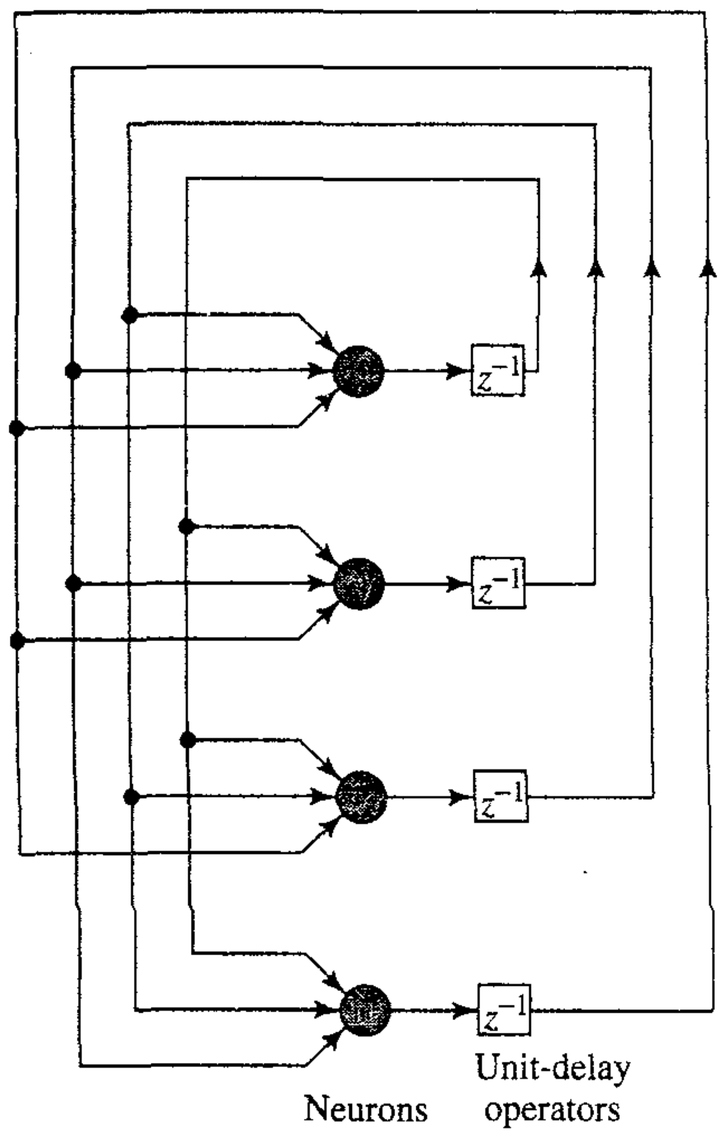


FIGURE 14.9 Architectural graph of a Hopfield network consisting of $N = 4$ neurons.

To study the dynamics of the Hopfield network, we use the neurodynamical model described in Eq. (14.16), which is based on the additive model of a neuron.

Recognizing that $x_i(t) = \varphi_i(v_i(t))$, we may rewrite Eq. (14.16) in the form

$$C_j \frac{d}{dt} v_j(t) = -\frac{v_j(t)}{R_j} + \sum_{i=1}^{N} w_{ji}\varphi_i(v_i(t)) + I_j, \qquad j = 1, \ldots, N \tag{14.20}$$

To proceed with the discussion, we make the following assumptions:

1. The matrix of synaptic weights is *symmetric*, as shown by

$$w_{ji} = w_{ij} \quad \text{for all } i \text{ and } j \tag{14.21}$$

2. Each neuron has a *nonlinear* activation of its own—hence the use of $\varphi_i(\cdot)$ in Eq. (14.20).
3. The *inverse* of the nonlinear activation function exists, so we may write

$$v = \varphi_i^{-1}(x) \tag{14.22}$$

Let the sigmoid function $\varphi_i(v)$ be defined by the hyperbolic tangent function

$$x = \varphi_i(v) = \tanh\left(\frac{a_i v}{2}\right) = \frac{1 - \exp(-a_i v)}{1 + \exp(-a_i v)} \tag{14.23}$$

which has a slope of $a_i/2$ at the origin as shown by

$$\frac{a_i}{2} = \left.\frac{d\varphi_i}{dv}\right|_{v=0} \tag{14.24}$$

Henceforth we refer to a_i as the *gain* of neuron i.

The inverse output–input relation of Eq. (14.22) may thus be rewritten in the form

$$v = \varphi_i^{-1}(x) = -\frac{1}{a_i}\log\left(\frac{1-x}{1+x}\right) \tag{14.25}$$

The *standard* form of the inverse output–input relation for a neuron of unity gain is defined by

$$\varphi^{-1}(x) = -\log\left(\frac{1-x}{1+x}\right) \tag{14.26}$$

We may rewrite Eq. (14.25) in terms of this standard relation as

$$\varphi_i^{-1}(x) = \frac{1}{a_i}\varphi^{-1}(x) \tag{14.27}$$

Figure 14.10a shows a plot of the standard sigmoidal nonlinearity $\varphi(v)$, and Fig. 14.10b shows the corresponding plot of the inverse nonlinearity $\varphi^{-1}(x)$.

The energy (Lyapunov) function of the Hopfield network in Fig. 14.9 is defined by (Hopfield, 1984)

$$E = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} w_{ji}x_i x_j + \sum_{j=1}^{N}\frac{1}{R_j}\int_0^{x_j}\varphi_j^{-1}(x)\,dx - \sum_{j=1}^{N} I_j x_j \tag{14.28}$$

The energy function E defined by Eq. (14.28) may have a complicated *landscape* with many minima. The dynamics of the network are described by a mechanism that seeks out those minima.

Hence, differentiating E with respect to time, we get

$$\frac{dE}{dt} = -\sum_{j=1}^{N}\left(\sum_{i=1}^{N} w_{ji}x_i - \frac{v_j}{R_j} + I_j\right)\frac{dx_j}{dt} \tag{14.29}$$

The quantity inside the parentheses on the right-hand side of Eq. (14.29) is recognized as $C_j\, dv_j/dt$ by virtue of the neurodynamical equation (14.20). We may thus simplify Eq. (14.29) to

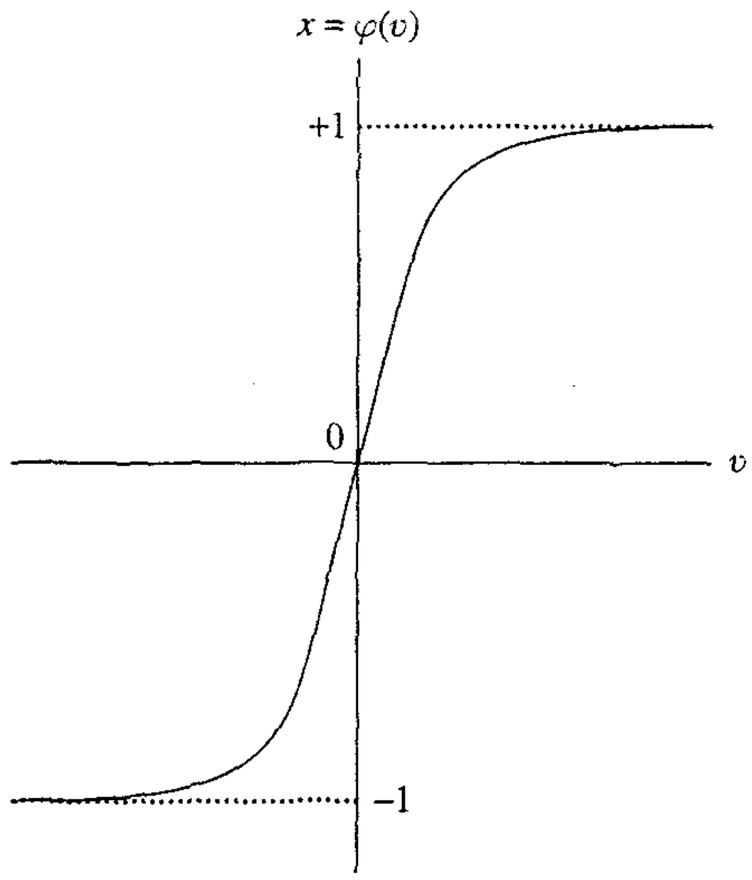
$$\frac{dE}{dt} = -\sum_{j=1}^{N} C_j\left(\frac{dv_j}{dt}\right)\frac{dx_j}{dt} \tag{14.30}$$

We now recognize the inverse relation that defines v_j in terms of x_j. The use of Eq. (14.22) in (14.30) yields

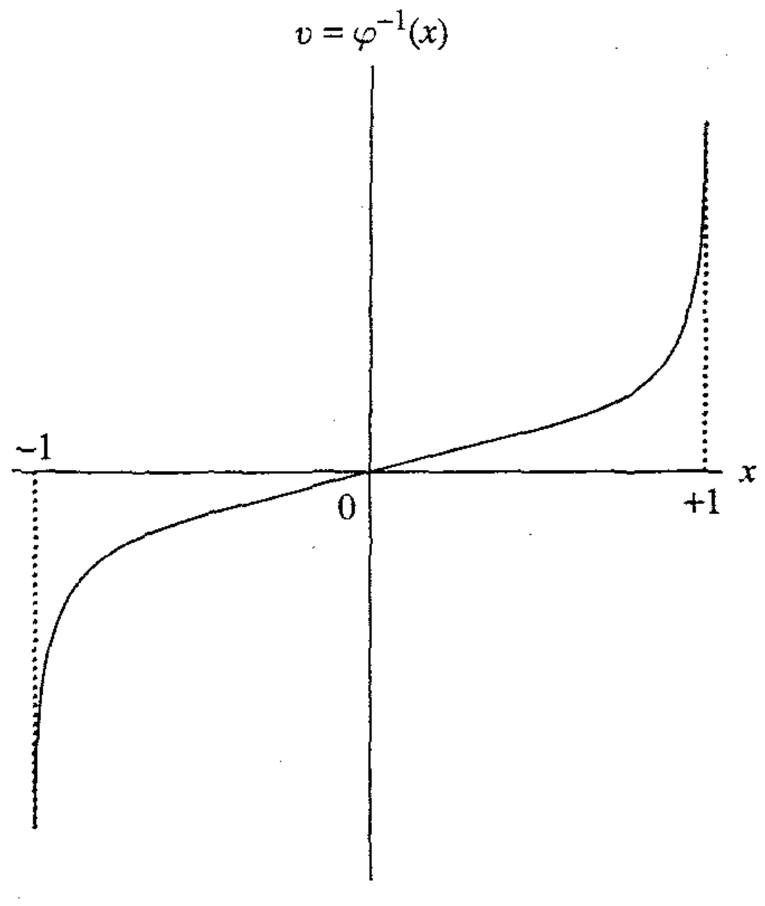
$$\begin{aligned}\frac{dE}{dt} &= -\sum_{j=1}^{N} C_j\left[\frac{d}{dt}\varphi_j^{-1}(x_j)\right]\frac{dx_j}{dt} \\ &= -\sum_{j=1}^{N} C_j\left(\frac{dx_j}{dt}\right)^2\left[\frac{d}{dx_j}\varphi_j^{-1}(x_j)\right]\end{aligned} \tag{14.31}$$

From Fig. 14.10b we see that the inverse output–input relation $\varphi_j^{-1}(x_j)$ is a monotonically increasing function of the output x_j. It follows therefore that

$$\frac{d}{dx_j}\varphi_j^{-1}(x_j) \geq 0 \quad \text{for all } x_j \tag{14.32}$$



(a)



(b)

FIGURE 14.10 Plots of (a) the standard sigmoidal nonlinearity, and (b) its inverse.

We also note that

$$\left(\frac{dx_j}{dt}\right)^2 \geq 0 \quad \text{for all } x_j \tag{14.33}$$

Hence, all the factors that make up the sum on the right-hand side of Eq. (14.31) are nonnegative. In other words, for the energy function E defined in Eq. (14.28), we have

$$\frac{dE}{dt} \leq 0$$

From the definition of Eq. (14.28), we note that the function E is bounded. Accordingly, we may make the following two statements:

1. The energy function E is a Lyapunov function of the continuous Hopfield model.
2. The model is stable in accordance with Lyapunov's Theorem 1.

In other words, the time evolution of the continuous Hopfield model described by the system of nonlinear first-order differential equations (14.20) represents a trajectory in state space, which seeks out the minima of the energy (Lyapunov) function E and comes to a stop at such fixed points. From Eq. (14.31) we also note that the derivative dE/dt vanishes only if

$$\frac{d}{dt}x_j(t) = 0 \quad \text{for all } j$$

We may thus go one step further and write

$$\frac{dE}{dt} < 0 \quad \text{except at a fixed point} \tag{14.34}$$

Equation (14.34) provides the basis for the following theorem:

The (Lyapunov) energy function E of a Hopfield network is a monotonically decreasing function of time .

Accordingly, the Hopfield network is globally asymptotically stable; the attractor fixed-points are the minima of the energy function, and vice versa.

Relation between the Stable States of the Discrete and Continuous Versions of the Hopfield Model

The Hopfield network may be operated in a continuous mode or discrete mode, depending on the model adopted for describing the neurons. The continuous mode of operation is based on an additive model, as previously described. On the other hand, the discrete mode of operation is based on the McCulloch–Pitts model. We may readily establish the relationship between the stable states of the continuous Hopfield model and those of the corresponding discrete Hopfield model by redefining the input–output relation for a neuron such that we may satisfy two simplifying characteristics:

1. The output of a neuron has the asymptotic values

$$x_j = \begin{cases} +1 & \text{for } v_j = \infty \\ -1 & \text{for } v_j = -\infty \end{cases} \tag{14.35}$$

2. The midpoint of the activation function of a neuron lies at the origin, as shown by

$$\varphi_j(0) = 0 \tag{14.36}$$

Correspondingly, we may set the bias I_j equal to zero for all j.

In formulating the energy function E for a continuous Hopfield model, the neurons are permitted to have self-loops. A discrete Hopfield model, on the other hand, need not have self-loops. We may therefore simplify our discussion by setting $w_{jj} = 0$ for all j in both models.

In light of these observations, we may redefine the energy function of a continuous Hopfield model given in Eq. (14.28) as follows:

$$E = -\frac{1}{2}\sum_{\substack{i=1\\ i\neq j}}^{N}\sum_{j=1}^{N} w_{ji}x_i x_j + \sum_{j=1}^{N}\frac{1}{R_j}\int_0^{x_j}\varphi_j^{-1}(x)dx \tag{14.37}$$

The inverse function $\varphi_j^{-1}(x)$ is defined by Eq. (14.27). We may thus rewrite the energy function of Eq. (14.37) as follows:

$$E = -\frac{1}{2}\sum_{\substack{i=1\\ i\neq j}}^{N}\sum_{j=1}^{N} w_{ji}x_i x_j + \sum_{j=1}^{N}\frac{1}{a_j R_j}\int_0^{x_j}\varphi^{-1}(x)dx \tag{14.38}$$

The integral

$$\int_0^{x_j}\varphi^{-1}(x)dx$$

has the standard form plotted in Fig. 14.11. Its value is zero for $x_j = 0$, and positive otherwise. It assumes a very large value as x_j approaches ± 1. If, however, the gain a_j of neuron j becomes infinitely large (i.e., the sigmoidal nonlinearity approaches the idealized hard-limiting form), the second term of Eq. (14.38) becomes negligibly small. In the limiting case when $a_j = \infty$ for all j, the maxima and minima of the continuous Hopfield model become identical with those of the corresponding discrete Hopfield model. In the latter case, the energy (Lyapunov) function is defined simply by

$$E = -\frac{1}{2}\sum_{\substack{i=1\\ i\neq j}}^{N}\sum_{j=1}^{N} w_{ji}x_i x_j \tag{14.39}$$

where the jth neuron state $x_j = \pm 1$. We conclude, therefore, that the only stable points of the very high-gain, continuous, deterministic Hopfield model correspond to the stable points of the discrete stochastic Hopfield model.

When, however, each neuron j has a large but finite gain a_j, we find that the second term on the right-hand side of Eq. (14.38) makes a noticeable contribution to the

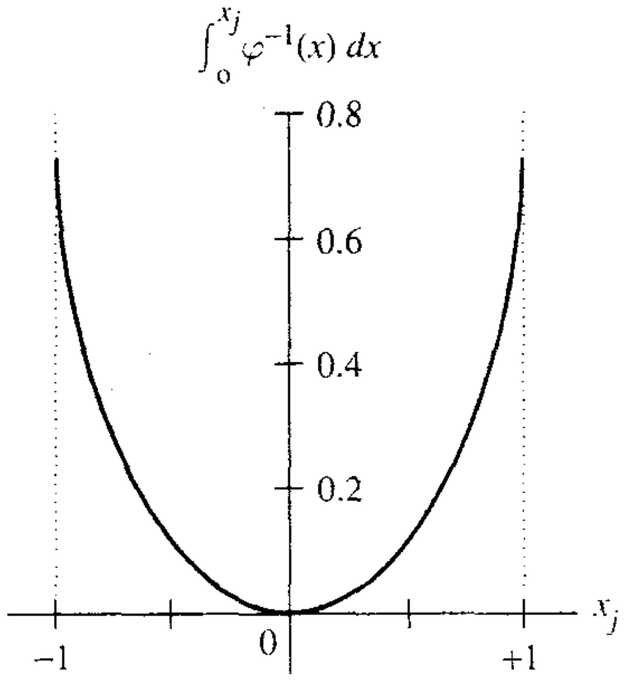


FIGURE 14.11 Plot of the integral $\int_0^{x_j}\varphi^{-1}(x)\,dx$.

energy function of the continuous model. In particular, this contribution is large and positive near all surfaces, edges, and corners of the unit hypercube that defines the state space of the model. On the other hand, the contribution is negligibly small at points that are far removed from the surface. Accordingly, the energy function of such a model has its maxima at corners, but the minima are displaced slightly toward the interior of the hypercube (Hopfield, 1984).

Figure 14.12 depicts the *energy contour map* or *energy landscape* for a continuous Hopfield model using two neurons. The outputs of the two neurons define the two axes of the map. The lower left- and upper right-hand corners of Fig. 14.12 represent stable minima for the limiting case of infinite gain; the minima for the case of finite gain are displaced inward. The flow to the fixed points (i.e., stable minima) may be interpreted as the solution to the minimization of the energy function E defined in Eq. (14.28).

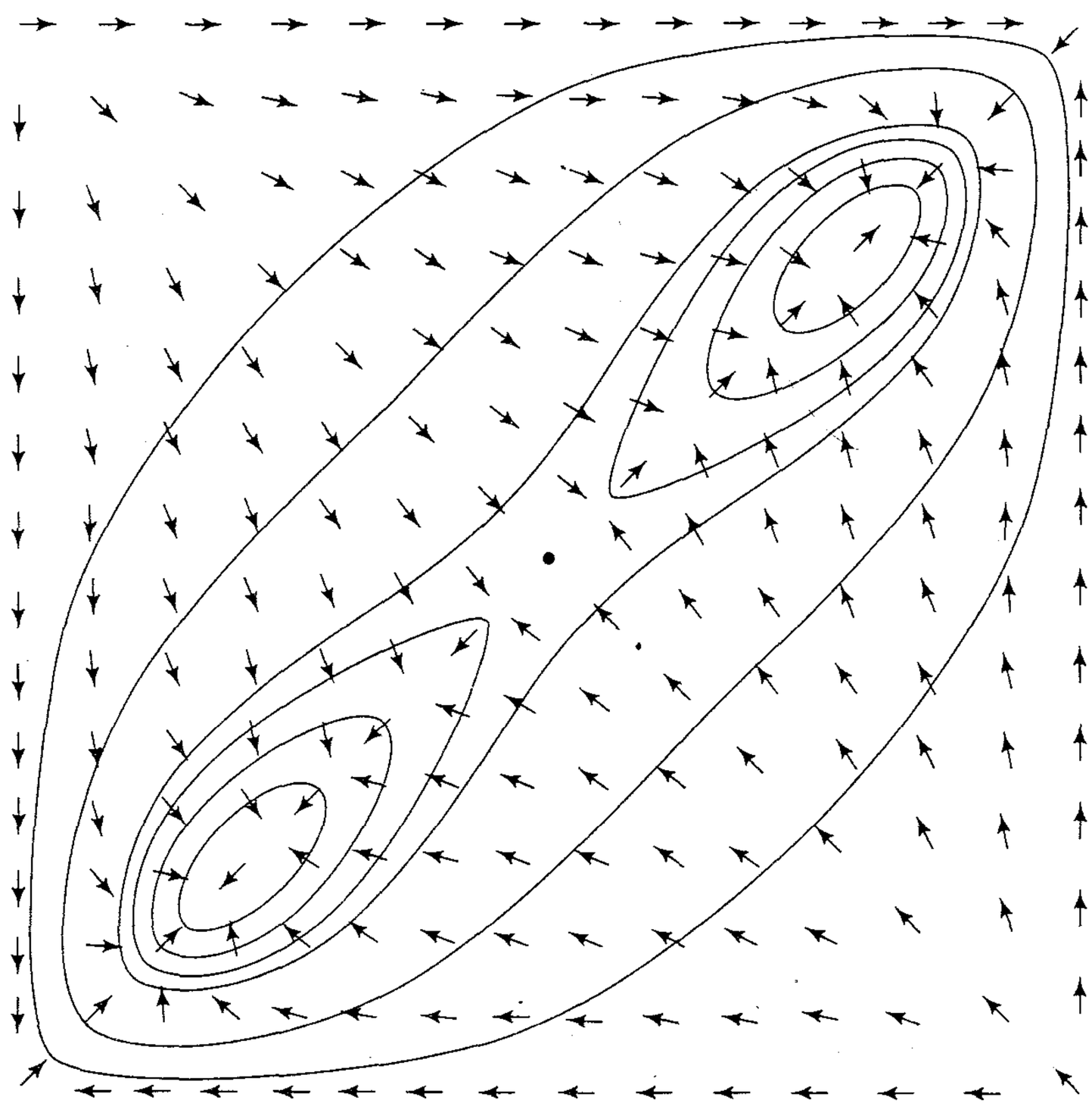


FIGURE 14.12 An energy contour map for a two-neuron, two-stable-state system. The ordinate and abscissa are the outputs of the two neurons. Stable states are located near the lower left and upper right corners, and unstable extrema at the other two corners. The arrows show the motion of the state. This motion is not generally perpendicular to the energy contours. (From J.J. Hopfield, 1984, with permission of the National Academy of Sciences of the U.S.A.)

The Discrete Hopfield Model as a Content-Addressable Memory

The Hopfield network has attracted a great deal of attention in the literature as a *content-addressable memory*. In this application, we know the fixed points of the network a *priori* in that they correspond to the patterns to be stored. However, the synaptic weights of the network that produce the desired fixed points are unknown, and the problem is how to determine them. The primary function of a content-addressable memory is to retrieve a pattern (item) stored in memory in response to the presentation of an incomplete or noisy version of that pattern. To illustrate the meaning of this statement in a succinct way, we can do no better than to quote from Hopfield's 1982 paper:

> Suppose that an item stored in memory is "H.A. Kramers & G.H. Wannier *Physi Rev. 60*, 252 (1941)." A general content-addressable memory would be capable of retrieving this entire memory item on the basis of sufficient partial information. The input "& Wannier (1941)" might suffice. An ideal memory could deal with errors and retrieve this reference even from the input "Wannier, (1941)."

An important property of a content-addressable memory is therefore the ability to retrieve a stored pattern, given a reasonable subset of the information content of that pattern. Moreover, a content-addressable memory is *error-correcting* in the sense that it can override inconsistent information in the cues presented to it.

The essence of a content-addressable memory (CAM) is to map a fundamental memory $\boldsymbol{\xi}_\mu$ onto a fixed (stable) point $\mathbf{x}_\mu$ of a dynamic system, as illustrated in Fig. 14.13. Mathematically, we may express this mapping in the form

$$\boldsymbol{\xi}_\mu \rightleftharpoons \mathbf{x}_\mu$$

The arrow from left to right describes an *encoding* operation, whereas the arrow from right to left describes a *decoding* operation. The attractor fixed points of the state space of the network are the *fundamental memories* or *prototype states* of the network.

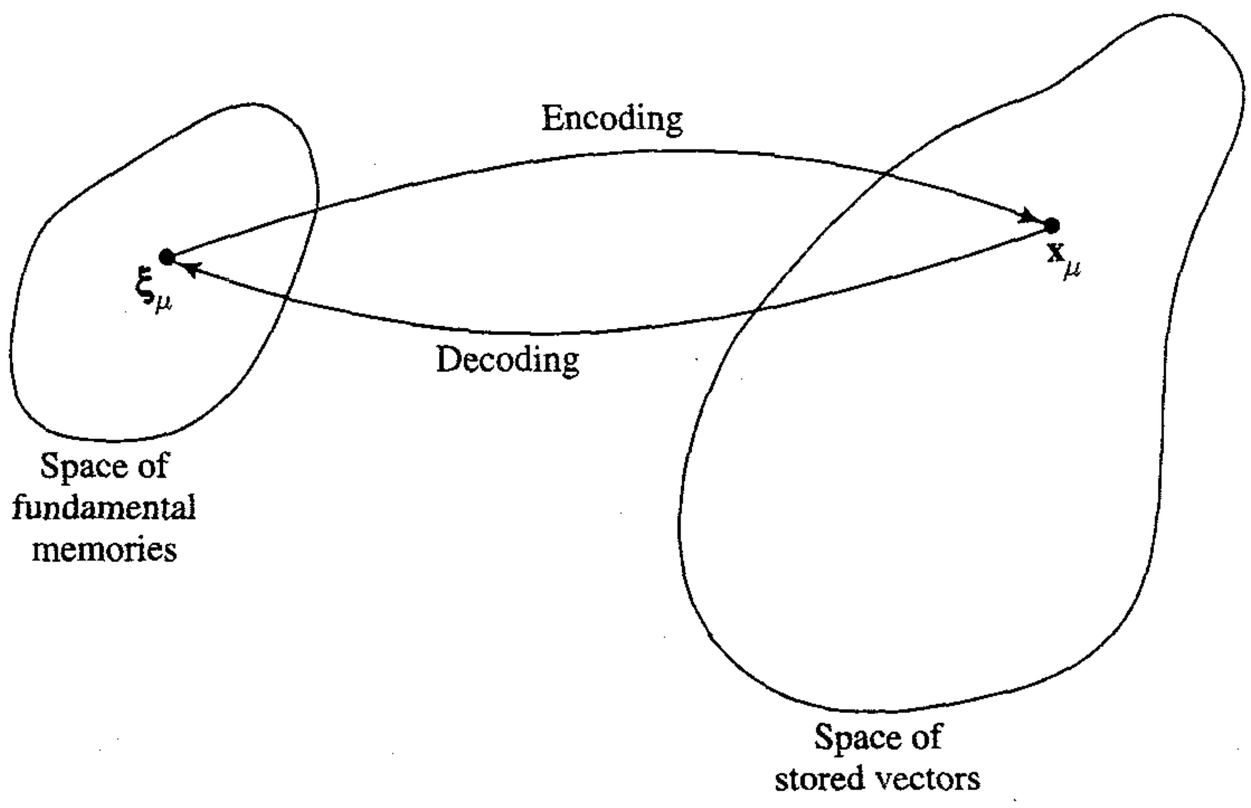


FIGURE 14.13 Illustration of the encoding–decoding performed by a recurrent network.

Suppose now that the network is presented a pattern containing partial but sufficient information about one of the fundamental memories. We may then represent that particular pattern as a starting point in the state space. In principle, provided that the starting point is close to the fixed point representing the memory being retrieved (i.e., it lies inside the basin of attraction belonging to the fixed point), the system should evolve with time and finally converge onto the memory state itself. At that point the entire memory is generated by the network. Consequently, the Hopfield network has an *emergent* property, which helps it retrieve information and cope with errors.

With the Hopfield model using the formal neuron of McCulloch and Pitts (1943) as its basic processing unit, each such neuron has two states determined by the level of the induced local field acting on it. The "on" or "firing" state of neuron i is denoted by the output $x_i = +1$, and the "off" or "quiescent" state is represented by $x_i = -1$. For a network made up of N neurons, the *state* of the network is thus defined by the vector

$$\mathbf{x} = [x_1, x_2, \ldots, x_N]^T$$

With $x_i = \pm 1$, the state of neuron i represents one *bit* of information, and the N-by-1 state vector $\mathbf{x}$ represents a binary word of N bits of information.

The induced local field v_j of neuron j is defined by

$$v_j = \sum_{i=1}^{N} w_{ji} x_i + b_j \tag{14.40}$$

where b_j is a fixed *bias* applied externally to neuron j. Hence, neuron j modifies its state x_j according to the *deterministic rule*

$$x_j = \begin{cases} +1 & \text{if } v_j > 0 \\ -1 & \text{if } v_j < 0 \end{cases}$$

This relation may be rewritten in the compact form

$$x_j = \operatorname{sgn}[v_j]$$

where sgn is the *signum function*. What if v_j is exactly zero? The action taken here can be quite arbitrary. For example, we may set $x_j = \pm 1$ if $v_j = 0$. However, we will use the following convention: If v_j is zero, neuron j remains in its previous state, regardless of whether it is on or off. The significance of this assumption is that the resulting flow diagram is symmetrical, as will be illustrated later.

There are two phases to the operation of the discrete Hopfield network as a content-addressable memory, namely the storage phase and the retrieval phase, as described here.

1. *Storage Phase.* Suppose that we wish to store a set of N-dimensional vectors (binary words), denoted by $\{\boldsymbol{\xi}_\mu | \mu = 1, 2, \ldots, M\}$. We call these M vectors *fundamental memories*, representing the patterns to be memorized by the network. Let $\xi_{\mu,i}$ denote the ith element of the fundamental memory $\boldsymbol{\xi}_\mu$, where the class $\mu = 1, 2, \ldots, M$. According to the *outer product rule* of storage, that is, the generalization of *Hebb's postulate of learning*, the synaptic weight from neuron i to neuron j is defined by

$$w_{ji} = \frac{1}{N} \sum_{\mu=1}^{M} \xi_{\mu,j} \xi_{\mu,i} \tag{14.41}$$

The reason for using $1/N$ as the constant of proportionality is to simplify the mathematical description of information retrieval. Note also that the learning rule of Eq. (14.41) is a "one shot" computation. In the normal operation of the Hopfield network, we set

$$w_{ii} = 0 \quad \text{for all } i \tag{14.42}$$

which means that the neurons have *no* self-feedback. Let $\mathbf{W}$ denote the N-by-N *synaptic weight matrix* of the network, with w_{ji} as its jith element. We may then combine Eqs. (14.41) and (14.42) into a single equation written in matrix form as follows:

$$\mathbf{W} = \frac{1}{N}\sum_{\mu=1}^{M} \boldsymbol{\xi}_\mu \boldsymbol{\xi}_\mu^T - M\mathbf{I} \tag{14.43}$$

where $\boldsymbol{\xi}_\mu \boldsymbol{\xi}_\mu^T$ represents the outer product of the vector $\boldsymbol{\xi}_\mu$ with itself, and $\mathbf{I}$ denotes the identity matrix. From these defining equations of the synaptic weights/weight matrix, we may reconfirm the following:

- The output of each neuron in the network is fed back to all other neurons.
- There is no self-feedback in the network (i.e., $w_{ii} = 0$).
- The weight matrix of the network is symmetric as shown by (see Eq. (14.21))

$$\mathbf{W}^T = \mathbf{W} \tag{14.44}$$

2. *Retrieval Phase.* During the retrieval phase, an N-dimensional vector $\boldsymbol{\xi}_{\text{probe}}$, called a *probe*, is imposed on the Hopfield network as its state. The probe vector has elements equal to ± 1. It typically represents an incomplete or noisy version of a fundamental memory of the network. Information retrieval then proceeds in accordance with a *dynamical rule* in which each neuron j of the network *randomly* but at some fixed rate examines the induced local field v_j (including any nonzero bias b_j) applied to it. If, at that instant of time, v_j is greater than zero, neuron j will switch its state to $+1$ or remain in that state if it is already there. Similarly, if v_j is less than zero, neuron j will switch its state to -1 or remain in that state if it is already there. If v_j is exactly zero, neuron j is left in its previous state, regardless of whether it is on or off. The state updating from one iteration to the next is therefore deterministic, but the selection of a neuron to perform the updating is done randomly. The *asynchronous* (serial) updating procedure described here is continued until there are no further changes to report. That is, starting with the probe vector $\mathbf{x}$, the network finally produces a time invariant state vector $\mathbf{y}$ whose individual elements satisfy the *condition for stability*:

$$y_i = \operatorname{sgn}\left(\sum_{i=1}^{N} w_{ji} y_i + b_j\right), \quad j = 1, 2, \ldots, N \tag{14.45}$$

or, in matrix form,

$$\mathbf{y} = \operatorname{sgn}(\mathbf{W}\mathbf{y} + \mathbf{b}) \tag{14.46}$$

where $\mathbf{W}$ is the synaptic weight matrix of the network, and $\mathbf{b}$ is the externally applied *bias vector*. The stability condition described here is also referred to as the *alignment condition*. The state vector $\mathbf{y}$ that satisfies it is called a *stable state* or *fixed point* of the state space of the system. We may therefore make the statement that the Hopfield network will always converge to a stable state when the retrieval operation is performed *asynchronously*.[5]

TABLE 14.2 Summary of the Hopfield Model

1. *Learning.* Let $\xi_1, \xi_2, \ldots, \xi_M$ denote a known set of N-dimensional fundamental memories. Use the outer product rule (i.e., Hebb's postulate of learning) to compute the synaptic weights of the network:

$$w_{ji} = \begin{cases} \dfrac{1}{N}\displaystyle\sum_{\mu=1}^{M} \xi_{\mu,j}\,\xi_{\mu,i}, & j \neq i \\ 0, & j = i \end{cases}$$

   where w_{ji} is the synaptic weight from neuron i to neuron j. The elements of the vector $\boldsymbol{\xi}_\mu$ equal ± 1. Once they are computed, the synaptic weights are kept fixed.
2. *Initialization.* Let $\boldsymbol{\xi}_{\text{probe}}$ denote an unknown N-dimensional input vector (probe) presented to the network. The algorithm is initialized by setting

$$x_j(0) = \xi_{j,\,\text{probe}}, \quad j = 1, \ldots, N$$

   where $x_j(0)$ is the state of neuron j at time $n = 0$, and $\xi_{j,\,\text{probe}}$ is the jth element of the probe vector $\boldsymbol{\xi}_{\text{probe}}$.
3. *Iteration Until Convergence.* Update the elements of state vector $\mathbf{x}(n)$ asynchronously (i.e., randomly and one at a time) according to the rule

$$x_j(n+1) = \text{sgn}\left[\sum_{i=1}^{N} w_{ji}x_i(n)\right], \quad j = 1, 2, \ldots, N$$

   Repeat the iteration until the state vector $\mathbf{x}$ remains unchanged.
4. *Outputting.* Let $\mathbf{x}_{\text{fixed}}$ denote the fixed point (stable state) computed at the end of step 3. The resulting output vector $\mathbf{y}$ of the network is

$$\mathbf{y} = \mathbf{x}_{\text{fixed}}$$

   Step 1 is the storage phase, and steps 2 through 4 constitute the retrieval phase.

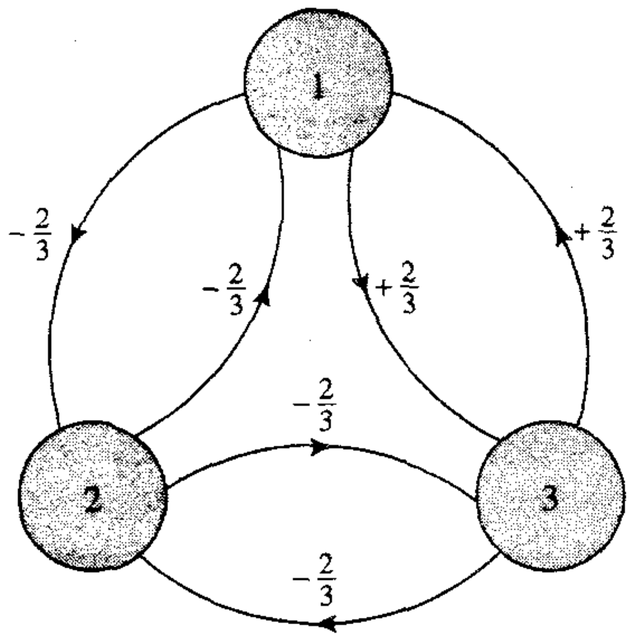
Table 14.2 presents a summary of the steps involved in the storage phase and retrieval phase of operating a Hopfield network.

Example 14.2

To illustrate the emergent behavior of the Hopfield model, consider the network of Fig. 14.14a, which consists of three neurons. The weight matrix of the network is

$$\mathbf{W} = \frac{1}{3}\begin{bmatrix} 0 & -2 & +2 \\ -2 & 0 & -2 \\ +2 & -2 & 0 \end{bmatrix}$$

which is legitimate since it satisfies the conditions of Eqs. (14.42) and (14.44). The bias applied to each neuron is assumed to be zero. With three neurons in the network, there are $2^3 = 8$ possible states to consider. Of these eight states, only the two states $(1, -1, 1)$ and $(-1, 1, -1)$ are stable; the remaining six states are all unstable. We say that these two particular states are stable because they both satisfy the alignment condition of Eq. (14.46). For the state vector $(1, -1, 1)$ we have



(a)

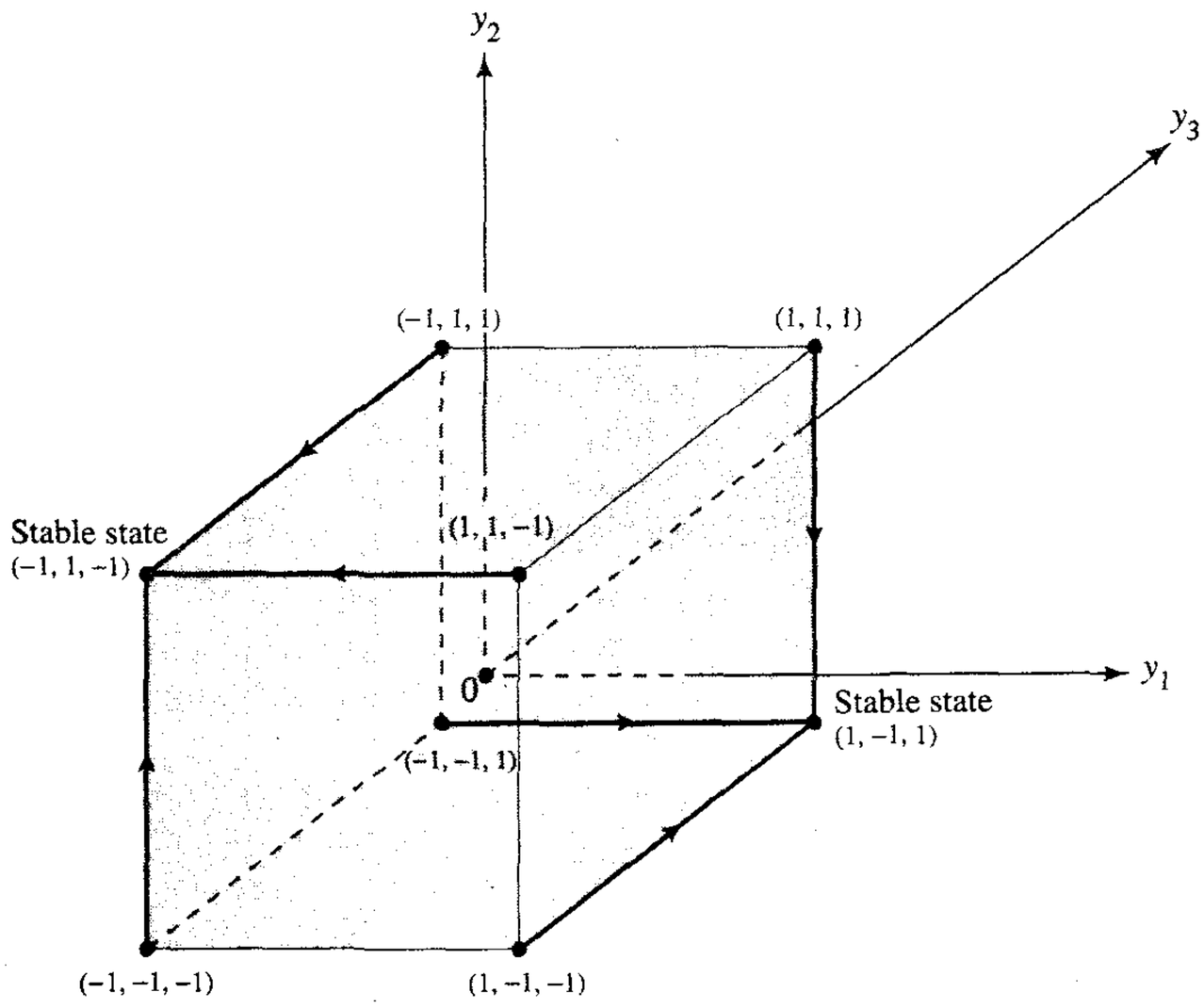


FIGURE 14.14
(a) Architectural graph of Hopfield network for $N = 3$ neurons. (b) Diagram depicting the two stable states and flow of the network.

$$\mathbf{Wy} = \frac{1}{3}\begin{bmatrix} 0 & -2 & +2 \\ -2 & 0 & -2 \\ +2 & -2 & 0 \end{bmatrix}\begin{bmatrix} +1 \\ -1 \\ +1 \end{bmatrix} = \frac{1}{3}\begin{bmatrix} +4 \\ -4 \\ +4 \end{bmatrix}$$

Hard limiting this result yields

$$\text{sgn}[\mathbf{Wy}] = \begin{bmatrix} +1 \\ -1 \\ +1 \end{bmatrix} = \mathbf{y}$$

Similarly, for the state vector $(-1, 1, -1)$ we have

$$\mathbf{Wy} = \frac{1}{3}\begin{bmatrix} 0 & -2 & +2 \\ -2 & 0 & -2 \\ +2 & -2 & 0 \end{bmatrix}\begin{bmatrix} -1 \\ +1 \\ -1 \end{bmatrix} = \frac{1}{3}\begin{bmatrix} -4 \\ +4 \\ -4 \end{bmatrix}$$

which, after hard limiting, yields

$$\text{sgn}[\mathbf{Wy}] = \begin{bmatrix} -1 \\ +1 \\ -1 \end{bmatrix} = \mathbf{y}$$

Hence, both of these state vectors satisfy the alignment condition.

Moreover, following the asynchronous updating procedure summarized in Table 14.2, we get the flow described in Fig. 14.14b. This flow map exhibits symmetry with respect to the two stable states of the network, which is intuitively satisfying. This symmetry is the result of leaving a neuron in its previous state if the induced local field acting on it is exactly zero.

Figure 14.14b also shows that if the network of Fig. 14.14a is in the initial state $(1, 1, 1)$, $(-1, -1, 1)$, or $(1, -1, -1)$, it will converge onto the stable state $(1, -1, 1)$ after one iteration. If the initial state is $(-1, -1, -1)$, $(-1, 1, 1)$, or $(1, 1, -1)$, it will converge onto the second stable state $(-1, 1, -1)$.

The network therefore has two fundamental memories, $(1, -1, 1)$ and $(-1, 1, -1)$, representing the two stable states. The application of Eq. (14.43) yields the synaptic weight matrix

$$\begin{aligned}\mathbf{W} &= \frac{1}{3}\begin{bmatrix} +1 \\ -1 \\ +1 \end{bmatrix}[+1, -1, +1] + \frac{1}{3}\begin{bmatrix} -1 \\ +1 \\ -1 \end{bmatrix}[-1, +1, -1] - \frac{2}{3}\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \frac{1}{3}\begin{bmatrix} 0 & -2 & +2 \\ -2 & 0 & -2 \\ +2 & -2 & 0 \end{bmatrix}\end{aligned}$$

which checks with the synaptic weights shown in Fig. 14.14a.

The error correcting capability of the Hopfield network is readily seen by examining the flow map of Fig. 14.14b:

1. If the probe vector $\boldsymbol{\xi}_{\text{probe}}$ applied to the network equals $(-1, -1, 1)$, $(1, 1, 1)$, or $(1, -1, -1)$, the resulting output is the fundamental memory $(1, -1, 1)$. Each of these values of the probe represents a single error, compared to the stored pattern.
2. If the probe vector $\boldsymbol{\xi}_{\text{probe}}$ equals $(1, 1, -1)$, $(-1, -1, -1)$, or $(-1, 1, 1)$, the resulting network output is the fundamental memory $(-1, 1, -1)$. Here again, each of these values of the probe represents a single error, compared to the stored pattern.

■

Spurious States

The weight matrix $\mathbf{W}$ of a discrete Hopfield network is symmetric, as indicated in Eq. (14.44). The eigenvalues of $\mathbf{W}$ are therefore all real. However, for large M, the eigenvalues are ordinarily *degenerate*, which means that there are several eigenvectors with the same eigenvalue. The eigenvectors associated with a degenerate eigenvalue form a subspace. Furthermore, the weight matrix $\mathbf{W}$ has a degenerate eigenvalue with a value of zero, in which case the subspace is called the *null space*. The null space exists by virtue of the fact that the number of fundamental memories, M, is smaller than the number of neurons, N, in the network. The presence of a null subspace is an intrinsic characteristic of the Hopfield network.

An eigenanalysis of the weight matrix $\mathbf{W}$ leads us to take the following viewpoint of the discrete Hopfield network used as a content-addressable memory (Aiyer et al., 1990):

1. The discrete Hopfield network acts as a *vector projector* in the sense that it projects a probe vector onto a subspace $\mathcal{M}$ spanned by the fundamental memory vectors.
2. The underlying dynamics of the network drive the resulting projected vector to one of the corners of a unit hypercube where the energy function is minimized.

The unit hypercube is N-dimensional. The M fundamental memory vectors, spanning the subspace $\mathcal{M}$, constitute a set of fixed points (stable states) represented by certain corners of the unit hypercube. The other corners of the unit hypercube that lie in or near subspace $\mathcal{M}$ are potential locations for *spurious states*, also referred to as *spurious attractors* (Amit, 1989). Spurious states represent stable states of the Hopfield network that are different from the fundamental memories of the network.

In the design of a Hopfield network as a content-addressable memory, we are therefore faced with a tradeoff between two conflicting requirements: (1) the need to preserve the fundamental memory vectors as fixed points in the state space, and (2) the desire to have few spurious states.

Storage Capacity of the Hopfield Network

Unfortunately, the fundamental memories of a Hopfield network are not always stable. Moreover, spurious states representing other stable states that are different from the fundamental memories can arise. These two phenomena tend to decrease the efficiency of the Hopfield network as a content-addressable memory. Here we explore the first of these two phenomena.

Let a probe equal to one of the fundamental memories, $\boldsymbol{\xi}_\nu$, be applied to the network. Then, permitting the use of self-feedback for generality and assuming zero bias, we find using Eq. (14.41) that the induced local field of neuron j is

$$\begin{aligned} v_j &= \sum_{i=1}^{N} w_{ji}\xi_{\nu,i} \\ &= \frac{1}{N}\sum_{\mu=1}^{M}\xi_{\mu,j}\sum_{i=1}^{N}\xi_{\mu,i}\xi_{\nu,i} \\ &= \xi_{\nu,j} + \frac{1}{N}\sum_{\substack{\mu=1\\ \mu\neq\nu}}^{M}\xi_{\mu,j}\sum_{i=1}^{N}\xi_{\mu,i}\xi_{\nu,i} \end{aligned} \tag{14.47}$$

The first term on the right-hand side of Eq. (14.47) is simply the jth element of the fundamental memory $\boldsymbol{\xi}_\nu$; now we can see why the scaling factor $1/N$ was introduced in the definition of the synaptic weight w_{ji} in Eq. (14.41). This term may therefore be viewed as the desired "signal" component of v_j. The second term on the right-hand side of Eq. (14.47) is the result of "crosstalk" between the elements of the fundamental memory $\boldsymbol{\xi}_\nu$ under test and those of some other fundamental memory $\boldsymbol{\xi}_\mu$. This second term may therefore be viewed as the "noise" component of v_j. We therefore have a situation similar to the classical "signal-in-noise detection problem" in communication theory (Haykin, 1994b).

We assume that the fundamental memories are random, being generated as a sequence of MN Bernoulli trials. The noise term of Eq. (14.47) then consists of a sum of

$N(M-1)$ independent random variables, taking values ± 1 divided by N. This is a situation where the central limit theorem of probability theory applies. The *central limit theorem* states (Feller, 1968):

> Let $\{X_k\}$ be a sequence of mutually independent random variables with a common distribution. Suppose that X_k has mean μ and variance σ^2, and let $Y = X_1 + X_2 + \cdots + X_n$ Then, as n approaches infinity, the probability distribution of the sum random variable Y approaches a Gaussian distribution.

Hence, by applying the central limit theorem to the noise term in Eq. (14.47), we find that the noise is asymptotically Gaussian distributed. Each of the $N(M-1)$ random variables constituting the noise term in this equation has a mean of zero and a variance of $1/N^2$. It follows, therefore, that the statistics of the Gaussian distribution are

- Zero mean
- Variance equal to $(M-1)/N$

The signal component $\xi_{v,j}$ has a value of $+1$ or -1 with equal probability, and therefore a mean of zero and variance of one. The *signal-to-noise ratio* is thus defined by

$$\begin{aligned}\rho &= \frac{\text{variance of signal}}{\text{variance of noise}} \\ &= \frac{1}{(M-1)/N} \\ &\simeq \frac{N}{M} \quad \text{for large } M\end{aligned} \tag{14.48}$$

The components of the fundamental memory $\boldsymbol{\xi}_v$ will be *stable* if, and only if, the signal-to-noise ratio ρ is high. Now, the number M of fundamental memories provides a direct measure of the *storage capacity* of the network. Therefore, it follows from Eq. (14.48) that so long as the storage capacity of the network is not overloaded—that is, the number M of fundamental memories is small compared to the number N of neurons in the network—the fundamental memories are stable in a probabilistic sense.

The reciprocal of the signal-to-noise ratio, that is,

$$\alpha = \frac{M}{N} \tag{14.49}$$

is called the *load parameter*. Statistical physics considerations reveal that the quality of memory recall of the Hopfield network deteriorates with increasing load parameter α, and breaks down at the *critical value* $\alpha_c = 0.14$ (Amit, 1989; Müller and Reinhardt, 1990). This critical value is in agreement with the estimate of Hopfield (1982), where it is reported that as a result of computer simulations $0.15N$ states can be recalled simultaneously before errors become severe.

With $\alpha_c = 0.14$, we find from Eq. (14.48) that the critical value of the signal-to-noise ratio is $\rho_c \simeq 7$, or equivalently 8.45 dB. For a signal-to-noise ratio below this critical value, memory recall breaks down.

The critical value

$$M_c = \alpha_c N = 0.14\, N \tag{14.50}$$

defines the *storage capacity with errors* on recall. To determine the storage capacity without errors we must use a more stringent criterion defined in terms of probability of error as described next.

Let the jth bit of the probe $\boldsymbol{\xi}_{\text{probe}} = \boldsymbol{\xi}_\nu$ be a symbol 1, that is, $\xi_{\nu,j} = 1$. Then the *conditional probability of bit error on recall* is defined by the shaded area in Fig. 14.15. The rest of the area under this curve is the *conditional probability that bit j of the probe is retrieved correctly*. Using the well-known formula for a Gaussian distribution, this latter conditional probability is given by

$$P(v_j > 0 | \xi_{\nu,j} = +1) = \frac{1}{\sqrt{2\pi}\sigma} \int_0^\infty \exp\left(-\frac{(v_j - \mu)^2}{2\sigma^2}\right) dv_j \tag{14.51}$$

With $\xi_{\nu,j}$ set to +1, and the mean of the noise term in Eq. (14.47) equal to zero, it follows that the mean of the random variable V is $\mu = 1$ and its variance is $\sigma^2 = (M - 1)/N$. From the definition of the *error function* commonly used in calculations involving the Gaussian distribution, we have

$$\text{erf}(y) = \frac{2}{\sqrt{\pi}} \int_0^y e^{-z^2}\, dz \tag{14.52}$$

where y is a variable defining the upper limit of integration. We may now simplify the expression for the conditional probability of correctly retrieving the jth bit of the fundamental memory $\boldsymbol{\xi}_\nu$ by rewriting Eq. (14.51) in terms of the error function as:

$$P(v_j > 0 | \xi_{\nu,j} = +1) = \frac{1}{2}\left[1 + \text{erf}\left(\sqrt{\frac{\rho}{2}}\right)\right] \tag{14.53}$$

where ρ is the signal-to-noise ratio defined in Eq. (14.48). Each fundamental memory consists of n bits. Also, the fundamental memories are usually equiprobable. It follows therefore that the *probability of stable patterns* is defined by

$$p_{\text{stab}} = (P(v_j > 0 | \xi_{\nu,j} = +1)^N \tag{14.54}$$

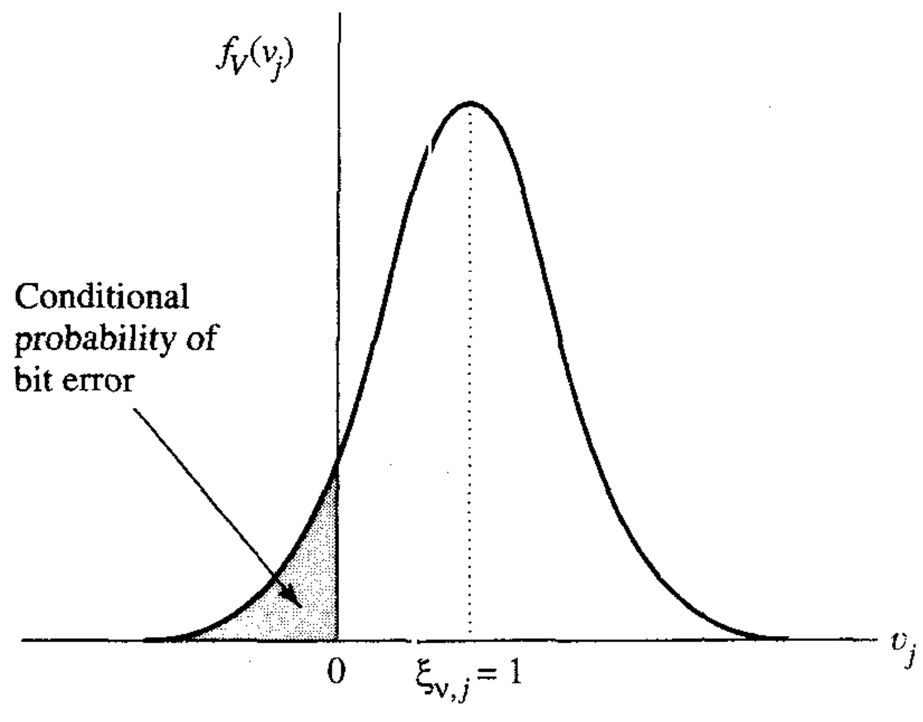


FIGURE 14.15 Conditional probability of bit error, assuming a Gaussian distribution for the induced local field v_j of neuron j; the subscript V in the probability density function $f_V(v_j)$ denotes a random variable with v_j representing a realization of it.

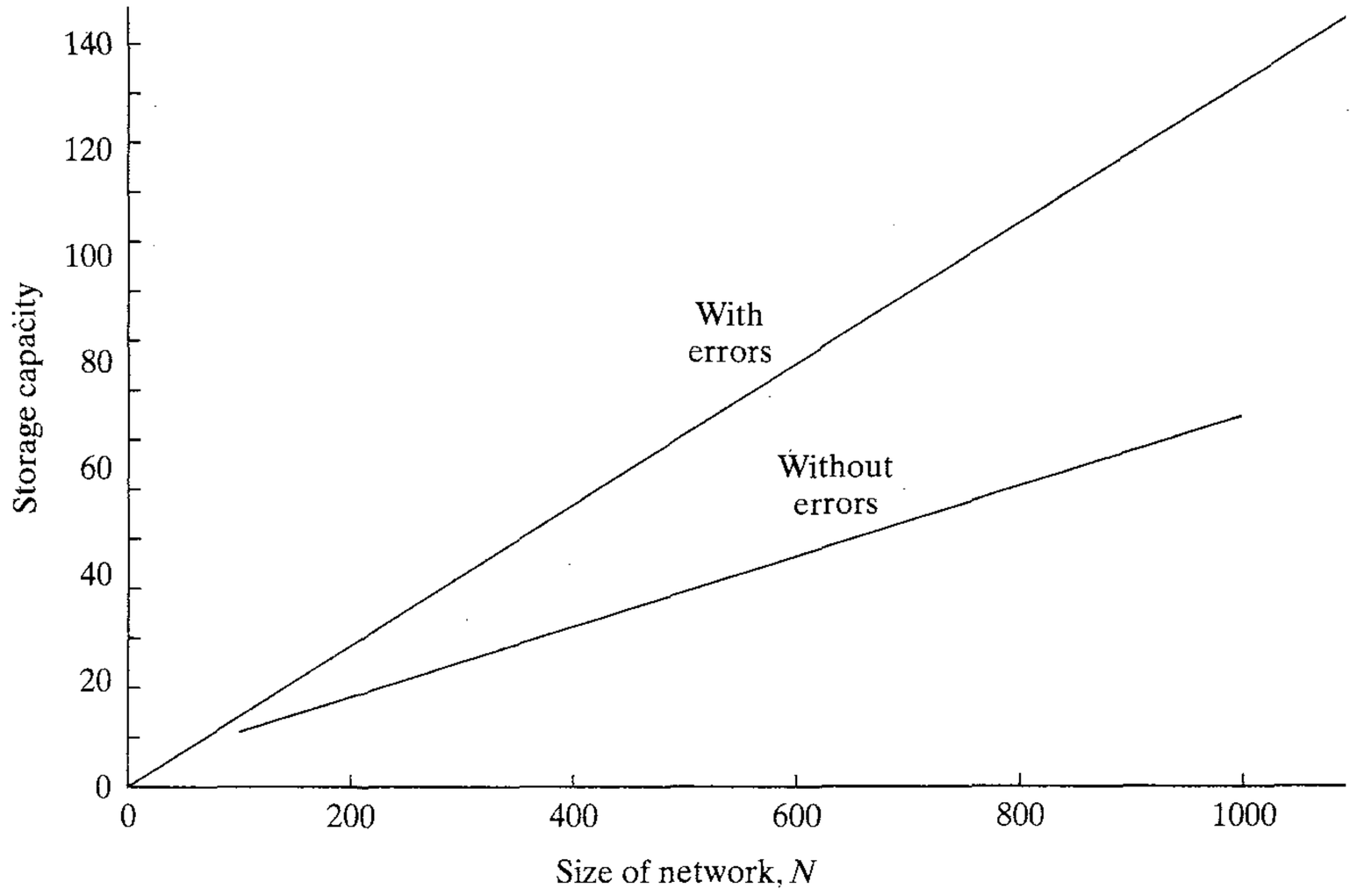


FIGURE 14.16 Plots of storage capacity of the Hopfield network versus network size for two cases: with errors and almost without errors.

We may use this probability to formulate an expression for the capacity of a Hopfield network. Specifically, we define the *storage capacity almost without errors*, $M_{\max}$, as the largest number of fundamental memories that can be stored in the network and yet insist that most of them be recalled correctly. In Problem 14.8 it is shown that this definition of storage capacity yields the formula

$$M_{\max} = \frac{N}{2 \log_e N} \tag{14.55}$$

where $\log_e$ denotes the natural logorithm.

Figure 14.16 shows graphs of the storage capacity with errors defined in Eq. (14.50) and the storage capacity almost without errors defined in Eq. (14.55), both plotted versus the network size N. From this figure we note the following points:

- Storage capacity of the Hopfield network scales essentially *linearly* with the size N of the network.
- A major limitation of the Hopfield network is that its storage capacity must be maintained small for the fundamental memories to be recoverable.[6]

14.8 COMPUTER EXPERIMENT I

In this section we use a computer experiment to illustrate the behavior of the discrete Hopfield network as a content-addressable memory. The network used in the experiment consists of $N = 120$ neurons, and therefore $N^2 - N = 12{,}280$ synaptic weights. It was

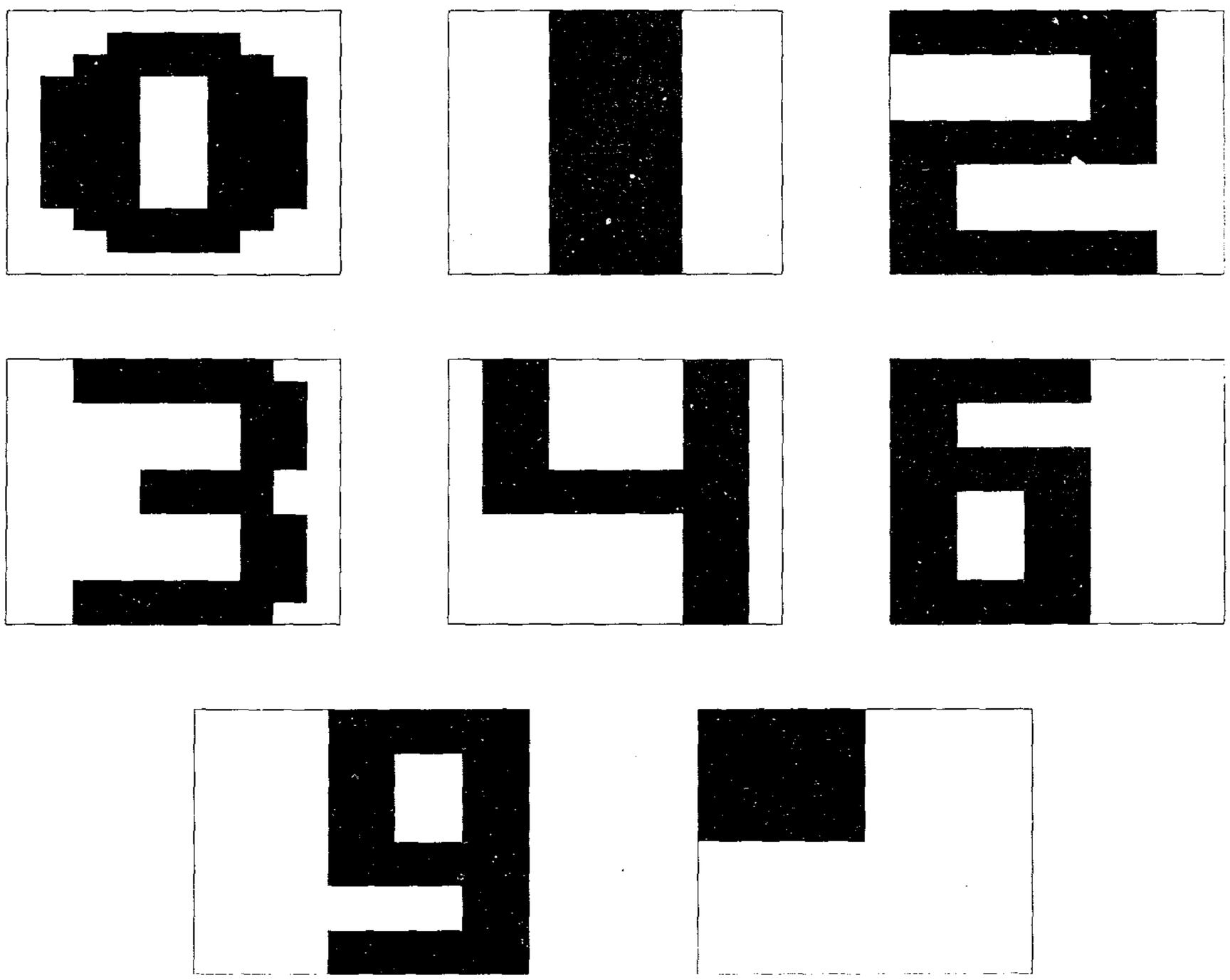


FIGURE 14.17 Set of handcrafted patterns for computer experiment on the Hopfield network.

trained to retrieve the eight digitlike black and white patterns shown in Fig. 14.17, with each pattern containing 120 pixels (picture elements) and designed specially to produce good performance (Lippmann, 1987). The inputs applied to the network assume the value $+1$ for black pixels and -1 for white pixels. The eight patterns of Fig. 14.17 were used as fundamental memories in the storage (learning) phase of the Hopfield network to create the synaptic weight matrix $\mathbf{W}$, which was done using Eq. (14.43). The retrieval phase of the network's operation was performed asynchronously, as described in Table 14.2.

During the first stage of the retrieval part of the experiment, the fundamental memories were presented to the network to test its ability to recover them correctly from the information stored in the synaptic weight matrix. In each case, the desired pattern was produced by the network after one iteration.

Next, to demonstrate the error-correcting capability of the Hopfield network, a pattern of interest was distorted by randomly and independently reversing each pixel of the pattern from $+1$ to -1 and vice versa with a probability of 0.25, and then using the corrupted pattern as a probe for the network. The result of this experiment for digit 3 is presented in Fig. 14.18. The pattern in the mid-top part of this figure represents a corrupted version of digit 3, which is applied to the network at zero time. The patterns produced by the network after 5, 10, 15, 20, 25, 30, and 35 iterations are presented in

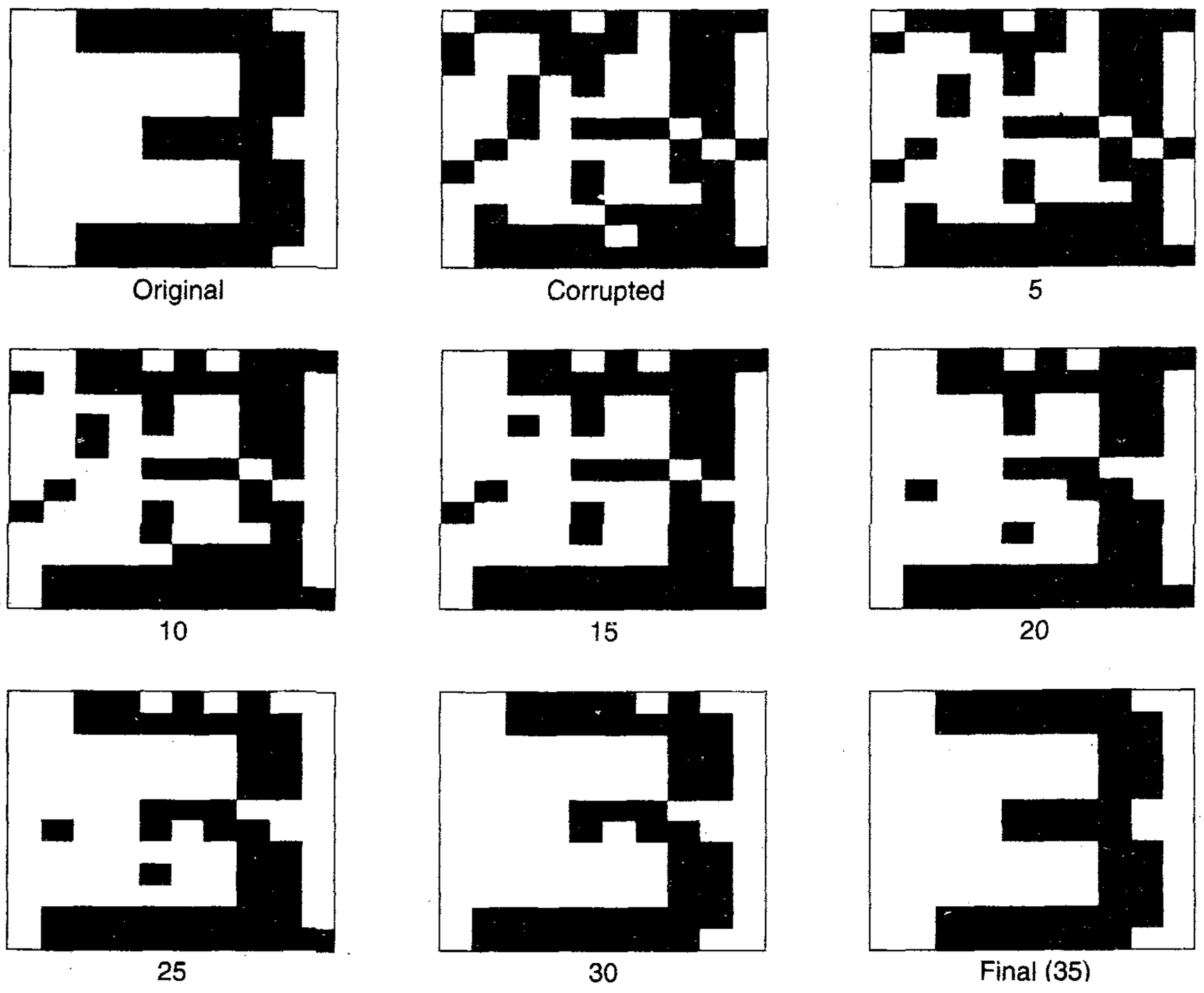


FIGURE 14.18 Correct recollection of corrupted pattern 3.

the rest of the figure. As the number of iterations is increased, we see that the resemblance of the network output to digit 3 is progressively improved. Indeed, after 35 iterations, the network converges onto the exactly correct form of digit 3.

Since, in theory, one quarter of the 120 neurons of the Hopfield network end up changing state for each corrupted pattern, the number of iterations needed for recall, on average, is 30. In our experiment, the number of iterations needed for the recall of the different patterns from their corrupted versions were as follows:

Pattern	Number of patterns needed for recall
0	34
1	32
2	26
3	35
4	25
6	37
"■"	32
9	26

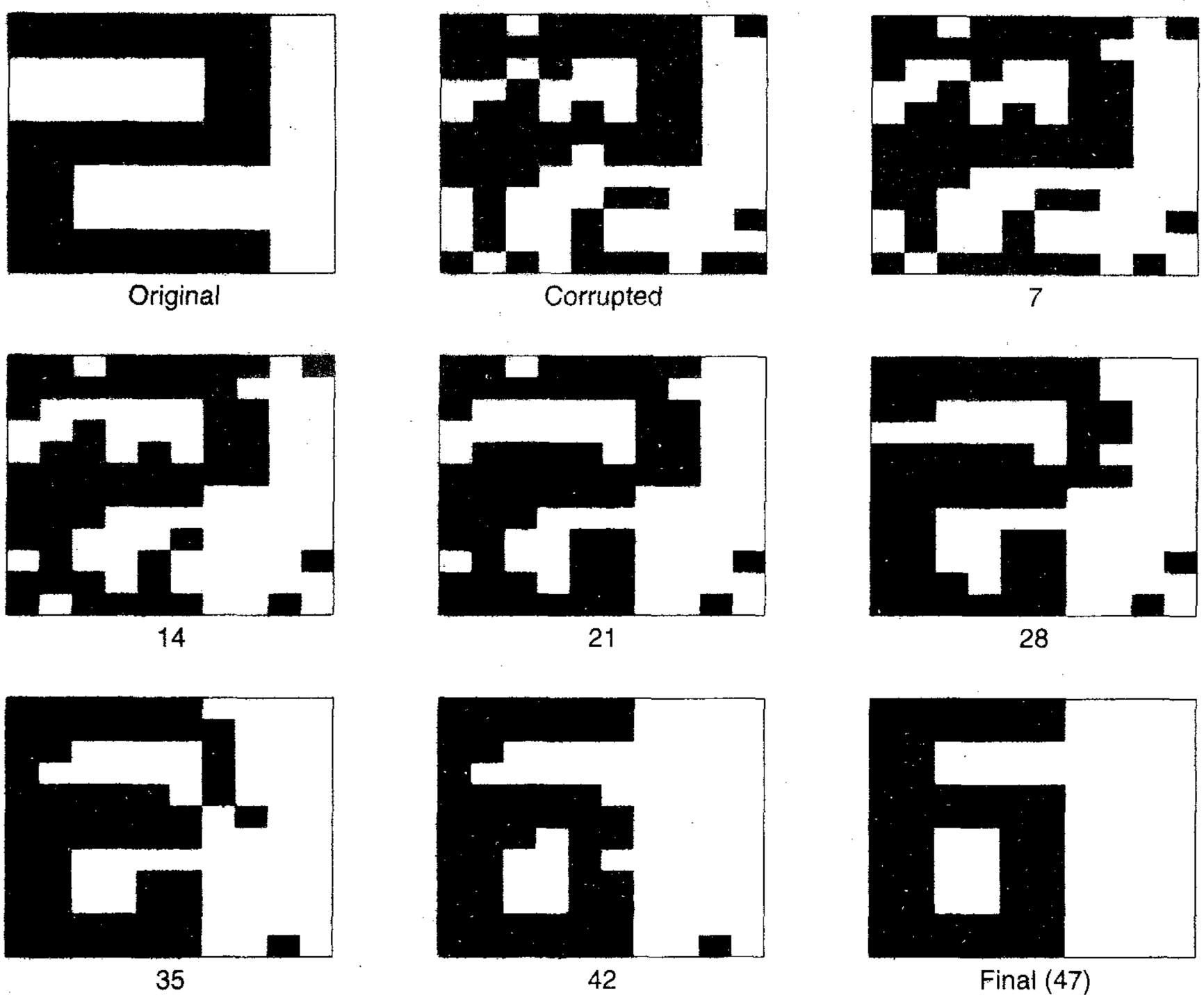


FIGURE 14.19 Incorrect recollection of corrupted pattern 2.

The average number of iterations needed for recall, averaged over the eight patterns, was about 31, which shows that the Hopfield network behaved as expected.

A problem inherent to the Hopfield network arises when the network is presented with a corrupted version of a fundamental memory, and the network then proceeds to converge onto the wrong fundamental memory. This is illustrated in Fig. 14.19, where the network is presented with a corrupted pattern "2," but after 47 iterations it converged to the fundamental memory "6."

As mentioned earlier, there is another problem that arises in the Hopfield network: the presence of spurious states. Figure 14.20 (viewed as a matrix of 14-by-8 network states) presents a listing of 108 spurious attractors found in 43,097 tests of randomly selected digits corrupted with the probability of flipping a bit set at 0.25. The spurious states may be grouped as follows (Amit, 1989):

1. *Reversed fundamental memories*. These spurious states are reversed (i.e., negative) versions of the fundamental memories of the network; see, for example, the state in location 1-by-1 in Fig. 14.20, which represents the negative of digit 6 in Fig. 14.17. To explain this kind of a spurious state, we note that the energy function E is symmetric in the sense that its value remains unchanged if the states of the neurons are reversed (i.e., the state x_i is replaced by $-x_i$ for all i).

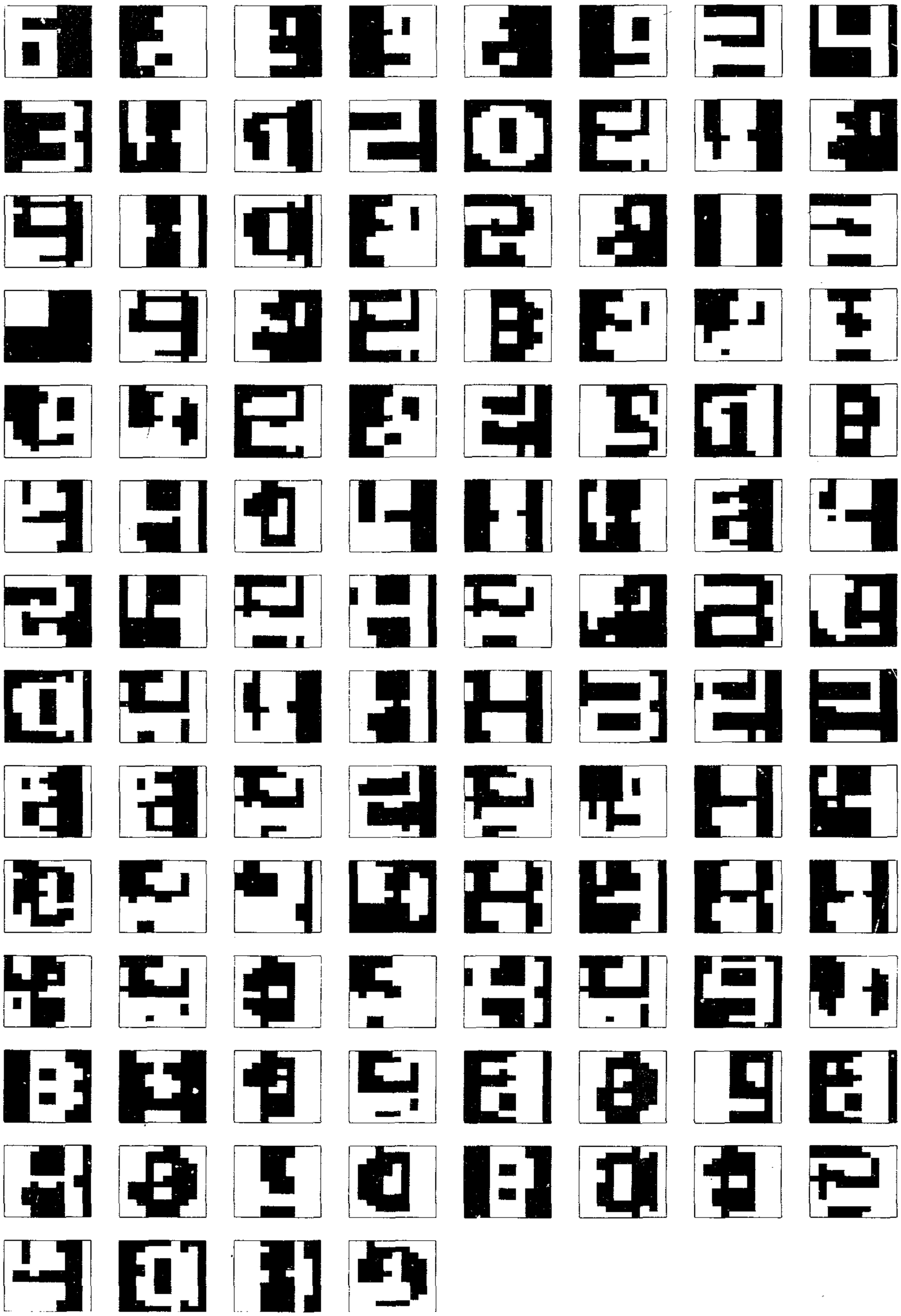


FIGURE 14.20 Compilation of the spurious states produced in the computer experiment on the Hopfield network.

Accordingly, if the fundamental memory $\boldsymbol{\xi}_\mu$ corresponds to a particular local minimum of the energy landscape, that same local minimum also corresponds to $-\boldsymbol{\xi}_\mu$. This sign reversal does not pose a problem in the retrieval of information if it is agreed to reverse all the information bits of a retrieved pattern if it is found that a particular bit designated as the "sign" bit is -1 instead of $+1$.

2. *Mixture states.* A mixture spurious state is a linear combination of an *odd* number of stored patterns. For example, consider the state

$$x_i = \text{sgn}(\xi_{1,i} + \xi_{2,i} + \xi_{3,i})$$

which is a three-mixture spurious state. It is a state formed out of three fundamental memories $\boldsymbol{\xi}_1$, $\boldsymbol{\xi}_2$, and $\boldsymbol{\xi}_3$ by a majority rule. The stability condition of Eq. (14.45) is satisfied by such a state for a large network. The state in location row 6, column 4 in Fig. 14.20 represents a three-mixture spurious state formed by a combination of the fundamental memories: $\boldsymbol{\xi}_1$ = negative of digit 1, $\boldsymbol{\xi}_2$ = digit 4, and $\boldsymbol{\xi}_3$ = digit 9.

3. *Spin-glass states.* This kind of a spurious state is so named by analogy with spin-glass models of statistical mechanics. Spin-glass states are defined by local minima of the energy landscape that are *not* correlated with any of the fundamental memories of the network; see, for example, the state in location row 7, column 6 in Fig. 14.20.

14.9 COHEN–GROSSBERG THEOREM

In Cohen and Grossberg (1983), a general principle for assessing the stability of a certain class of neural networks is described by the following system of coupled nonlinear differential equations:

$$\frac{d}{dt}u_j = a_j(u_j)\left[b_j(u_j) - \sum_{i=1}^{N} c_{ji}\varphi_i(u_i) \right], \quad j = 1, \ldots, N \tag{14.56}$$

According to Cohen and Grossberg, this class of neural networks admits a Lyapunov function defined as

$$E = \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} c_{ji}\varphi_i(u_i)\varphi_j(u_j) - \sum_{j=1}^{N}\int_0^{u_j} b_j(\lambda)\varphi_j'(\lambda)d\lambda \tag{14.57}$$

where

$$\varphi_j'(\lambda) = \frac{d}{d\lambda}(\varphi_j(\lambda)) \tag{14.58}$$

For the definition of Eq. (14.57) to be valid, however, we require the following conditions to hold:

1. The synaptic weights of the network are "symmetric:"

$$c_{ij} = c_{ji} \tag{14.59}$$

2. The function $a_j(u_j)$ satisfies the condition for "nonnegativity:"

$$a_j(u_j) \geq 0 \tag{14.60}$$

To derive the relation of Eq. (11.27), we may manipulate the summation on the right-hand side of this equation as follows:

$$\sum_{i=1}^{K} \pi_i p_{ij} = \sum_{i=1}^{K} \left(\frac{\pi_i}{\pi_j} p_{ij} \right) \pi_j$$

$$= \sum_{i=1}^{K} (p_{ji}) \pi_j$$

$$= \pi_j$$

In the second line of this expression we made use of the principle of detailed balance, and in the last line we made use of the fact that the transition probabilities of a Markov chain satisfy the condition (see Eq. (11.15) with the roles of i and j interchanged):

$$\sum_{i=1}^{K} p_{ji} = 1 \quad \text{for all } j$$

Note that the principle of detailed balance implies that the distribution $\{\pi_j\}$ is a stationary distribution.

11.4 METROPOLIS ALGORITHM

Now that we understand the composition of a Markov chain, we will use it to formulate a stochastic algorithm for simulating the evolution of a physical system to thermal equilibrium. The algorithm is called the *Metropolis algorithm* (Metropolis et al., 1953). It is a modified Monte Carlo method, introduced in the early days of scientific computation for the stochastic simulation of a collection of atoms in equilibrium at a given temperature.

Suppose that the random variable X_n representing an arbitrary Markov chain is in state x_i at time n. We randomly generate a new state x_j, representing a realization of another random variable Y_n. It is assumed that the generation of this new state satisfies the symmetry condition:

$$P(Y_n = x_j | X_n = x_i) = P(Y_n = x_i | X_n = x_j)$$

Let ΔE denote the energy difference resulting from the transition of the system from state $X_n = x_i$ to state $Y_n = x_j$. If the energy difference ΔE is negative, the transition leads to a state with lower energy and the transition is accepted. The new state is then accepted as the starting point for the next step of the algorithm, that is, we put $X_{n+1} = Y_n$. If, on the other hand, the energy difference ΔE is positive, the algorithm proceeds in a probabilistic manner at that point. First, we select a random number ξ uniformly distributed in the range $[0, 1]$. If $\xi < \exp(-\Delta E/T)$, where T is the operating temperature, the transition is accepted and we put $X_{n+1} = Y_n$. Otherwise, the transition is rejected and we put $X_{n+1} = X_n$; that is, the old configuration is reused for the next step of the algorithm.

Choice of Transition Probabilities

Let the arbitrary Markov chain have *a priori* transition probabilities denoted by τ_{ij}, which satisfy three conditions:

1. *Nonnegativity*:

$$\tau_{ij} \geq 0 \quad \text{for all } (i, j)$$

2. *Normalization*:

$$\sum_j \tau_{ij} = 1 \quad \text{for all } i$$

3. *Symmetry*:

$$\tau_{ij} = \tau_{ji} \quad \text{for all } (i, j)$$

Let π_i denote the steady state probability that the Markov chain is in state x_i, $i = 1, 2, \ldots, K$. We may then use the symmetric τ_{ij}s and the probability distribution ratio π_j/π_i, to be defined, to formulate the desired set of transition probabilities as (Beckerman, 1997):

$$p_{ij} = \begin{cases} \tau_{ij}\left(\dfrac{\pi_j}{\pi_i}\right) & \text{for } \dfrac{\pi_j}{\pi_i} < 1 \\ \tau_{ij} & \text{for } \dfrac{\pi_j}{\pi_i} \geq 1 \end{cases} \tag{11.29}$$

To ensure that the transition probabilities are normalized to unity, we introduce this additional definition for the probability of no transition:

$$p_{ii} = \tau_{ii} + \sum_{j \neq i} \tau_{ij}\left(1 - \frac{\pi_j}{\pi_i}\right) \tag{11.30}$$

$$= 1 - \sum_{j \neq i} \alpha_{ij}\tau_{ij}$$

where α_{ij} is the moving probability defined by

$$\alpha_{ij} = \min\left(1, \frac{\pi_j}{\pi_i}\right) \tag{11.31}$$

The only outstanding requirement is how to choose the ratio π_j/π_i. To cater to this requirement, we choose the probability distribution that we want the Markov chain to converge to be a Gibbs distribution, as shown by

$$\pi_j = \frac{1}{Z}\exp\left(-\frac{E_j}{T}\right)$$

in which case the probability distribution ratio π_j/π_i takes the simple form

$$\frac{\pi_j}{\pi_i} = \exp\left(-\frac{\Delta E}{T}\right) \tag{11.32}$$

where

$$\Delta E = E_j - E_i \tag{11.33}$$

By using the ratio of probability distributions we have eliminated dependence on the partition function Z.

By construction, the transition probabilities are all nonnegative and normalized to unity, as required by Eqs. (11.14) and (11.15). Moreover, they satisfy the principle of detailed balance defined by Eq. (11.28). This principle is a sufficient condition for thermal equilibrium. To demonstrate that the principle of detailed balance is satisfied, we offer the following considerations:

Case 1: $\Delta E < 0$**.** Suppose that in going from state x_i to state x_j, the energy change ΔE is negative. From Eq. (11.32) we find that $(\pi_j/\pi_i) > 1$, so the use of Eq. (11.29) yields

$$\pi_i p_{ij} = \pi_i \tau_{ij} = \pi_i \tau_{ji}$$

and

$$\pi_j p_{ji} = \pi_j \left(\frac{\pi_i}{\pi_j} \tau_{ji} \right) = \pi_i \tau_{ji}$$

Hence, the principle of detailed balance is satisfied for $\Delta E < 0$.

Case 2: $\Delta E > 0$**.** Suppose next that the energy change ΔE in going from state x_i to state x_j is positive. In this case we find that $(\pi_j/\pi_i) < 1$, and the use of Eq. (11.29) yields

$$\pi_i p_{ij} = \pi_i \left(\frac{\pi_j}{\pi_i} \tau_{ij} \right) = \pi_j \tau_{ij} = \pi_j \tau_{ji}$$

and

$$\pi_j p_{ji} = \pi_j \tau_{ji}$$

Here again we see that the principle of detailed balance is satisfied.

To complete the picture, we need to clarify the use of the *a priori* transition probabilities denoted by τ_{ij}. These transition probabilities are in fact the probabilistic model of the random step in the Metropolis algorithm. From the description of the algorithm presented earlier, we recall that the random step is followed by a random decision. We may therefore conclude that the transition probabilities p_{ij} defined in Eqs. (11.29) and (11.30) in terms of the *a priori* transition probabilities, τ_{ij}, and the steady state probabilities, π_j, are indeed the correct choice for the Metropolis algorithm.

It is noteworthy that the stationary distribution generated by the Metropolis algorithm does not uniquely determine the Markov chain. The Gibbs distribution at equilibrium may be generated by using an update rule other than the Monte Carlo rule applied in the Metropolis algorithm. For example, it may be generated using the Boltzmann learning rule due to Ackley et al. (1986); this latter rule is discussed in Section 11.7.

11.5 SIMULATED ANNEALING

Consider the problem of finding a low energy system whose states are ordered in a Markov chain. From Eq. (11.11) we observe that as the temperature T approaches zero. the free energy F of the system approaches the average energy $<E>$. With $F \rightarrow <E>$. we next observe from the principle of minimal free energy that the Gibbs distribution. which is the stationary distribution of the Markov chain, collapses on the global minima

of the average energy $<E>$ as $T \rightarrow 0$. In other words, low energy ordered states are strongly favored at low temperatures. These observations prompt us to raise the question: Why not simply apply the Metropolis algorithm for generating a population of configurations representative of the stochastic system at very low temperatures? We do not advocate the use of such a strategy because the rate of convergence of the Markov chain to thermal equilibrium is extremely slow at very low temperatures. Rather, the preferred method for improved computational efficiency is to operate the stochastic system at a high temperature where convergence to equilibrium is fast, and then maintain the system at equilibrium as the temperature is carefully lowered. That is, we use a combination of two related ingredients:

- A schedule that determines the rate at which the temperature is lowered.
- An algorithm—exemplified by the Metropolis algorithm—that iteratively finds the equilibrium distribution at each new temperature in the schedule by using the final state of the system at the previous temperature as the starting point for the new temperature.

The twofold scheme that we have just described is the essence of a widely used stochastic relaxation technique known as *simulated annealing*[2] (Kirkpatrick et al., 1983). The technique derives its name from analogy with an annealing process in physics/chemistry where we start the process at high temperature and then lower the temperature slowly while maintaining thermal equilibrium.

The primary objective of simulated annealing is to find the global minimum of a cost function that characterizes large and complex systems.[3] As such, it provides a powerful tool for solving nonconvex optimization problems, motivated by the following simple idea:

> *When optimizing a very large and complex system (i.e., a system with many degrees of freedom), instead of always going downhill, try to go downhill most of the time.*

Simulated annealing differs from conventional iterative optimization algorithms in two important respects:

- The algorithm need not get stuck, since transition out of a local minimum is always possible when the system operates at a nonzero temperature.
- Simulated annealing is *adaptive* in that gross features of the final state of the system are seen at higher temperatures, while fine details of the state appear at lower temperatures.

Annealing Schedule

As already mentioned, the Metropolis algorithm is the basis for the simulated annealing process, in the course of which the temperature T is decreased slowly. That is, the temperature T plays the role of a *control* parameter. The simulated annealing process will converge to a configuration of minimal energy provided that the temperature is decreased no faster than logarithmically. Unfortunately, such an annealing schedule is extremely slow—too slow to be of practical use. In practice, we must resort to a *finite-time approximation* of the asymptotic convergence of the algorithm. The price paid for

the approximation is that the algorithm is no longer guaranteed to find a global minimum with probability 1. Nevertheless, the resulting approximate form of the algorithm is capable of producing near optimum solutions for many practical applications.

To implement a finite-time approximation of the simulated annealing algorithm. we must specify a set of parameters governing the convergence of the algorithm. These parameters are combined in a so-called *annealing schedule* or *cooling schedule*. The annealing schedule specifies a finite sequence of values of the temperature and a finite number of transitions attempted at each value of the temperature. The annealing schedule due to Kirkpatrick et al. (1983) specifies the parameters of interest as follows:[4]

- *Initial Value of the Temperature.* The initial value T_0 of the temperature is chosen high enough to ensure that virtually all proposed transitions are accepted by the simulated annealing algorithm
- *Decrement of the Temperature.* Ordinarily, the cooling is performed *exponentially*, and the changes made in the value of the temperature are small. In particular, the *decrement function* is defined by

$$T_k = \alpha T_{k-1}, \quad k = 1, 2, \ldots \tag{11.34}$$

  where α is a constant smaller than, but close to, unity. Typical values of α lie between 0.8 and 0.99. At each temperature, enough transitions are attempted so that there are 10 *accepted* transitions per experiment on the average.
- *Final Value of the Temperature.* The system is frozen and annealing stops if the desired number of acceptances is not achieved at three successive temperatures.

The latter criterion may be refined by requiring that the *acceptance ratio*, defined as the number of accepted transitions divided by the number of proposed transitions, is smaller than a prescribed value (Johnson et al., 1989).

Simulated Annealing for Combinatorial Optimization

Simulated annealing is particularly well suited for solving combinatorial optimization problems. The objective of *combinatorial optimization* is to minimize the cost function of a finite, discrete system characterized by a large number of possible solutions. Essentially, simulated annealing uses the Metropolis algorithm to generate a sequence of solutions by invoking an analogy between a physical many-particle system and a combinatorial optimization problem.

In simulated annealing, we interpret the energy E_i in the Gibbs distribution of Eq. (11.5) as a numerical cost and the temperature T as a control parameter. The numerical cost assigns to each configuration in the combinatorial optimization problem a scalar value that describes how desirable that particular configuration is to the solution. The next issue in the simulated annealing procedure to be considered is how to identify configurations and generate new configurations from previous ones in a local manner. This is where the Metropolis algorithm performs its role. We may thus summarize the correspondence between the terminology of statistical physics and that of combinatorial optimization as shown in Table 11.1 (Beckerman, 1997).

11.6 GIBBS SAMPLING

Like the Metropolis algorithm, the *Gibbs sampler*[5] generates a Markov chain with the Gibbs distribution as the equilibrium distribution. However, the transition probabilities associated with the Gibbs sampler are nonstationary (Geman and Geman, 1984). In the final analysis, the choice between the Gibbs sampler and the Metropolis algorithm is based on technical details of the problem at hand.

To proceed with a description of this sampling scheme, consider a K-dimensional random vector $\mathbf{X}$ made up of the components $X_1, X_2, \ldots, X_K$. Suppose that we have knowledge of the conditional distribution of X_k, given values of all the other components of $\mathbf{X}$ for $k = 1, 2, \ldots, K$. The problem we wish to address is how to obtain a numerical estimate of the marginal density of the random variable X_k for each k. The Gibbs sampler proceeds by generating a value for the conditional distribution for each component of the random vector $\mathbf{X}$, given the values of all other components of $\mathbf{X}$. Specifically, starting from an arbitrary configuration $\{x_1(0), x_2(0), \ldots, x_K(0)\}$, we make the following drawings on the first iteration of Gibbs sampling:

$x_1(1)$ is drawn from the distribution of X_1, given $x_2(0), x_3(0), \ldots, x_K(0)$.
$x_2(1)$ is drawn from the distribution of X_2, given $x_1(1), x_3(0), \ldots, x_K(0)$.
$\vdots$
$x_k(1)$ is drawn from the distribution of X_k, given $x_1(1), \ldots, x_{k-1}(1), x_{k+1}(0), \ldots, x_K(0)$.
$\vdots$
$x_K(1)$ is drawn from the distribution of X_K, given $x_1(1), x_2(1), \ldots, x_{K-1}(1)$.

We proceed in this same manner on the second iteration and every other iteration of the sampling scheme. The following two points should be carefully noted:

1. Each component of the random vector $\mathbf{X}$ is "visited" in the natural order, with the result that a total of K new variates are generated on each iteration.
2. The new value of component X_{k-1} is used immediately when a new value of X_k is drawn for $k = 2, 3, \ldots, K$.

From this discussion we see that the Gibbs sampler is an *iterative adaptive* scheme. After n iterations of its use, we arrive at the K variates: $X_1(n), X_2(n), \ldots, X_K(n)$.

TABLE 11.1 Correspondence between Statistical Physics and Combinatorial Optimization

Statistical physics	Combinatorial optimization
Sample	Problem instance
State (configuration)	Configuration
Energy	Cost function
Temperature	Control parameter
Ground-state energy	Minimum cost
Ground-state configuration	Optimal configuration

Under mild conditions, the following three theorems hold for Gibbs sampling (Geman and Geman, 1984; Gelfand and Smith, 1990):

1. *Convergence theorem. The random variable $X_k(n)$ converges in distribution to the true probability distributions of X_k for $k = 1, 2, \ldots, K$ as n approaches infinity*; that is,

$$\lim_{n\to\infty} P(X_k^{(n)} \leq x | x_k(0)) = F_{X_k}(x) \quad \text{for } k = 1, 2, \ldots, K \tag{11.35}$$

where $F_{X_k}(x)$ is the marginal probability distribution function of X_k.

In fact, a stronger result is proven in Geman and Geman (1984). Specifically, rather than requiring that each component of the random vector **X** be visited in repetitions of the natural order, convergence of Gibbs sampling still holds under an arbitrary visiting scheme provided that this scheme does not depend on the values of the variables and that each component of **X** is visited on an "infinitely often" basis.

2. *Rate of convergence theorem. The joint probability distribution of the random variables $X_1(n), X_2(n), \ldots, X_K(n)$ converges to the true joint probability distribution of $X_1, X_2, \ldots, X_K$ at a geometric rate in n.*

This theorem assumes that the components of **X** are visited in the natural order. When, however, an arbitrary but infinitely often visiting approach is used, then a minor adjustment to the rate of convergence is required.

3. *Ergodic theorem. For any measurable function g, for example, of the random variables $X_1, X_2, \ldots, X_K$ whose expectation exists, we have*

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} g(X_1(i), X_2(i), \ldots, X_K(i)) \to E[g(X_1, X_2, \ldots, X_K)] \tag{11.36}$$

with probability 1 *(i.e., almost surely).*

The ergodic theorem tells us how to use the output of the Gibbs sampler to obtain numerical estimations of the desired marginal densities.

Gibbs sampling is used in the Boltzmann machine to sample from distributions over hidden neurons; this stochastic machine is discussed in the next section. In the context of a stochastic machine using binary units (e.g., Boltzmann machine), it is noteworthy that the Gibbs sampler is exactly the same as a variant of the Metropolis algorithm. In the standard form of the Metropolis algorithm, we go downhill with probability 1. In contrast, in the alternative form of the Metropolis algorithm, we go downhill with a probability equal to 1 minus the exponential of the energy gap (i.e., the complement of the uphill rule). In other words, if a change lowers the energy E or leaves it unchanged, that change is accepted; if the change increases the energy, it is accepted with probability $\exp(-\Delta E)$ and is rejected otherwise, with the old state then being repeated (Neal, 1993).

11.7 BOLTZMANN MACHINE

The *Boltzmann machine* is a stochastic machine whose composition consists of stochastic neurons. A *stochastic neuron* resides in one of two possible states in a probabilistic manner, as discussed in Chapter 1. These two states may be designated as +1

for the "on" state and −1 for the "off" state, or 1 and 0, respectively. We will adopt the former designation. Another distinguishing feature of the Boltzmann machine is the use of *symmetric synaptic connections* between its neurons. The use of this form of synaptic connections is also motivated by statistical physics considerations.

The stochastic neurons of the Boltzmann machine partition into two functional groups: *visible* and *hidden*, as depicted in Fig. 11.4. The visible neurons[6] provide an interface between the network and the environment in which it operates. During the training phase of the network, the visible neurons are all *clamped* onto specific states determined by the environment. The hidden neurons, on the other hand, always operate freely; they are used to explain underlying constraints contained in the environmental input vectors. The hidden neurons accomplish this task by capturing higher-order statistical correlations in the clamping vectors. The network described here represents a special case of the Boltzmann machine. It may be viewed as an unsupervised learning procedure for modeling a probability distribution that is specified by clamping patterns onto the visible neurons with appropriate probabilities. By so doing, the network can perform *pattern completion*. Specifically, when a partial information-bearing vector is clamped onto a subset of the visible neurons, the network performs completion on the remaining visible neurons, provided that it has learned the training distribution properly (Hinton, 1989).

The primary goal of Boltzmann learning is to produce a neural network that correctly models input patterns according to a Boltzmann distribution. In applying this form of learning, two assumptions are made:

- Each environmental input vector (pattern) persists long enough to permit the network to reach *thermal equilibrium*.
- There is *no* structure in the sequential order in which the environmental vectors are clamped onto the visible units of the network.

A particular set of synaptic weights is said to constitute a perfect model of the environmental structure if it leads to exactly the same probability distribution of the states of

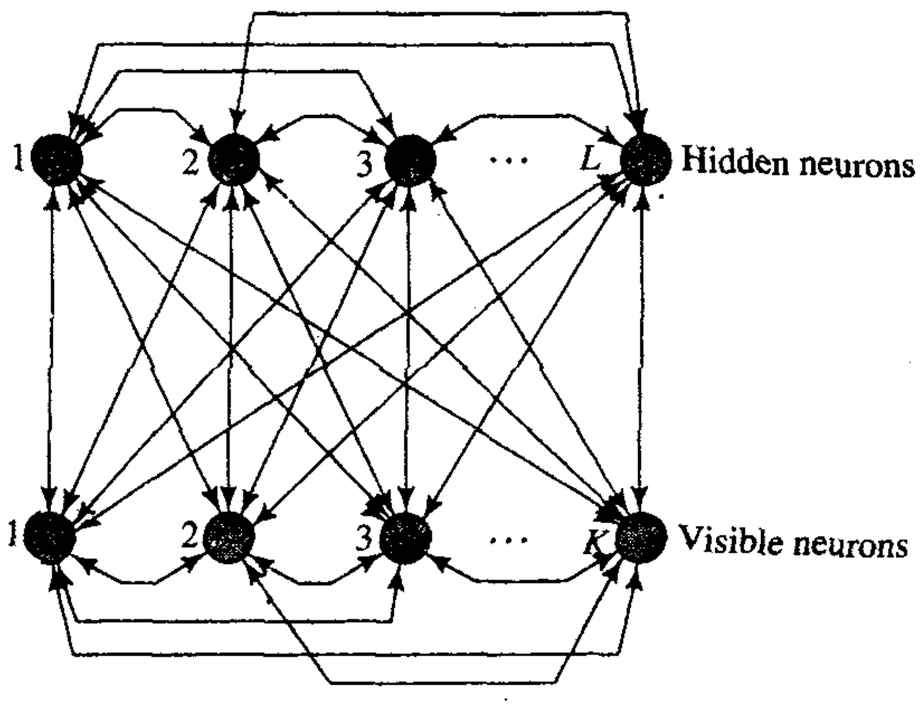


FIGURE 11.4 Architectural graph of Botzmann machine; K is the number of visible neurons and L is the number of hidden neurons.

the visible units (when the network is running freely) as when these units are clamped by the environmental input vectors. In general, unless the number of hidden units is exponentially large compared to the number of visible units, it is impossible to achieve such a perfect model. If, however, the environment has a regular structure, and the network uses its hidden units to capture these regularities, it may achieve a good match to the environment with a manageable number of hidden units.

Gibbs Sampling and Simulated Annealing for the Boltzmann Machine

Let $\mathbf{x}$ denote the state vector of the Boltzmann machine, with its component x_i denoting the state of neuron i. The state $\mathbf{x}$ represents a realization of the random vector $\mathbf{X}$. The synaptic connection from neuron i to neuron j is denoted by w_{ji}, with

$$w_{ji} = w_{ij} \quad \text{for all } (i, j) \tag{11.37}$$

and

$$w_{ii} = 0 \quad \text{for all } i \tag{11.38}$$

Equation (11.37) describes symmetry and Eq. (11.38) emphasizes the absence of self-feedback. The use of a bias is permitted by using the weight w_{j0} from a fictitious node maintained at $+1$ and by connecting it to neuron j for all j.

From an analogy with thermodynamics, the energy of the Boltzmann machine is defined by[7]

$$E(\mathbf{x}) = -\frac{1}{2}\sum_{\substack{i \\ i \neq j}}\sum_{j} w_{ji} x_i x_j \tag{11.39}$$

Invoking the Gibbs distribution of Eq. (11.5), we may define the probability that the network (assumed to be in equilibrium at temperature T) is in state $\mathbf{x}$ as follows:

$$P(\mathbf{X} = \mathbf{x}) = \frac{1}{Z}\exp\left(-\frac{E(\mathbf{x})}{T}\right) \tag{11.40}$$

where Z is the partition function.

To simplify the presentation, define the single event A and joint events B and C as follows:

$$A: X_j = x_j$$
$$B: \{X_i = x_i\}_{i=1, i \neq j}^{K}$$
$$C: \{X_i = x_i\}_{i=1}^{K}$$

In effect, the joint event B excludes A, and the joint event C includes both A and B. The probability of B is the marginal probability of C with respect to A. Hence, using Eqs. (11.39) and (11.40), we may write

$$\begin{aligned} P(C) &= (A, B) \\ &= \frac{1}{Z}\exp\left(\frac{1}{2T}\sum_{\substack{i \\ i \neq j}}\sum_{j} w_{ji} x_i x_j\right) \end{aligned} \tag{11.41}$$

and

$$\begin{aligned} P(B) &= \sum_{A} P(A, B) \\ &= \frac{1}{Z} \sum_{x_j} \exp\left(\frac{1}{2T} \sum_{\substack{i \\ i \neq j}} \sum_{j} w_{ji} x_i x_j\right) \end{aligned} \tag{11.42}$$

The exponent in Eqs. (11.41) and (11.42) may be expressed as the sum of two components, one involving x_j and the other being independent of x_j. The component involving x_j is given by

$$\frac{x_j}{2T} \sum_{\substack{i \\ i \neq j}} w_{ji} x_i$$

Accordingly, by setting $x_j = x = \pm 1$, we may express the conditional probability of A, given B, as follows:

$$\begin{aligned} P(A|B) &= \frac{P(A, B)}{P(B)} \\ &= \frac{1}{1 + \exp\left(-\frac{x_j}{T} \sum_{\substack{i \\ i \neq j}} w_{ji} x_i\right)} \end{aligned}$$

That is, we may write

$$P(X_j = x | \{X_i = x_i\}_{i=1, i \neq j}^{K}) = \varphi\left(\frac{x}{T} \sum_{\substack{i \\ i \neq j}} w_{ji} x_i\right) \tag{11.43}$$

where $\varphi(\cdot)$ is a sigmoid function of its argument, as shown by

$$\varphi(v) = \frac{1}{1 + \exp(-v)} \tag{11.44}$$

Note that although x varies between -1 and $+1$, the whole argument $v = \frac{x}{T} \sum_{i \neq j} w_{ji} x_i$ for large N may vary between $-\infty$ and $+\infty$, as depicted in Fig. 11.5. Note also, in deriving

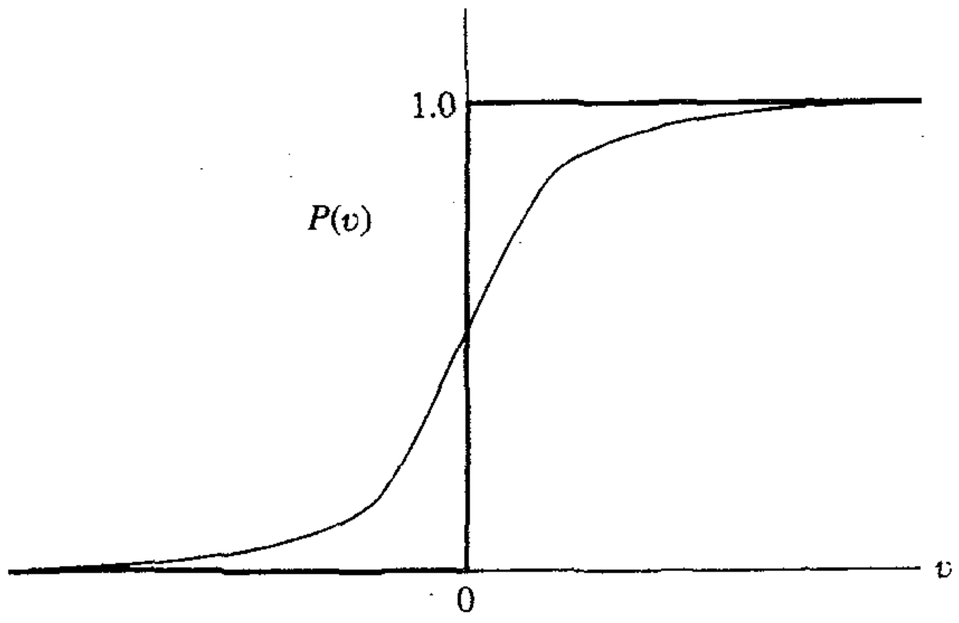


FIGURE 11.5 Sigmoid-shaped function $\varphi(v)$.

Eq. (11.43), the need for the partition function Z has been eliminated. This is highly desirable since a direct computation of Z is infeasible for a network of large complexity.

The use of Gibbs sampling exhibits the joint distribution $P(A, B)$. Basically, as explained in Section 11.6, this stochastic simulation starts with the network assigned an arbitrary state, and the neurons are all repeatedly visited in their natural order. On each visit, a new value for the state of each neuron is chosen in accordance with the probability distribution for that neuron, conditional on the values for the states of all other neurons in the network. Provided that the stochastic simulation is performed long enough, the network will reach thermal equilibrium at temperature T.

Unfortunately, the time taken to reach thermal equilibrium can be much too long. To overcome this difficulty, simulated annealing for a finite sequence of temperatures T_0, $T_1, \ldots, T_{\text{final}}$ is used, as explained in Section 11.5. Specifically, the temperature is initially set to the high value T_0, thereby permitting thermal equilibrium to be reached fast. Thereafter, the temperature T is gradually reduced to the final value T_{final}, at which point the neuronal states will have (hopefully) reached their desired marginal distributions.

Boltzmann Learning Rule

Because the Boltzmann machine is a stochastic machine, it is natural to look to probability theory for an appropriate index of performance. One such criterion is the *likelihood function*.[8] On this basis, the goal of Boltzmann learning is to maximize the likelihood function or, equivalently, the log-likelihood function, in accordance with the *maximum-likelihood principle*.

Let $\mathcal{T}$ denote the set of training examples drawn from the probability distribution of interest. It is assumed that the examples are all two-valued. Repetition of training examples is permitted in proportion to how common certain cases are known to occur. Let a subset of the state vector $\mathbf{x}$, say $\mathbf{x}_\alpha$, denote the state of the visible neurons. The remaining part of the state vector $\mathbf{x}$, say $\mathbf{x}_\beta$, represents the state of the hidden neurons. The state vectors $\mathbf{x}$, $\mathbf{x}_\alpha$ and $\mathbf{x}_\beta$ are realizations of the random vectors $\mathbf{X}$, $\mathbf{X}_\alpha$, and $\mathbf{X}_\beta$, respectively. There are two phases to the operation of the Boltzmann machine:

- *Positive phase*. In this phase the network operates in its clamped condition (i.e., under the direct influence of the training set $\mathcal{T}$).
- *Negative phase*. In this second phase, the network is allowed to run freely, and therefore with no environmental input.

Given the synaptic weight vector $\mathbf{w}$ for the whole network, the probability that the visible neurons are in state $\mathbf{x}_\alpha$ is $P(\mathbf{X}_\alpha = \mathbf{x}_\alpha)$. With the many possible values of $\mathbf{x}_\alpha$ contained in the training set $\mathcal{T}$, assumed to be statistically independent, the overall probability distribution is the factorial distribution $\prod_{\mathbf{x}_\alpha \in \mathcal{T}} P(\mathbf{X}_\alpha = \mathbf{x}_\alpha)$. To formulate the log-likelihood function $L(\mathbf{w})$, take the logarithm of this factorial distribution and treat $\mathbf{w}$ as the unknown parameter vector. We may thus write

$$\begin{aligned} L(\mathbf{w}) &= \log \prod_{\mathbf{x}_\alpha \in \mathcal{T}} P(\mathbf{X}_\alpha = \mathbf{x}_\alpha) \\ &= \sum_{\mathbf{x}_\alpha \in \mathcal{T}} \log P(\mathbf{X}_\alpha = \mathbf{x}_\alpha) \end{aligned} \tag{11.45}$$

To formulate the expression for the marginal probability $P(\mathbf{X}_\alpha = \mathbf{x}_\alpha)$ in terms of the energy function $E(\mathbf{x})$, we use the following:

- The probability $P(\mathbf{X} = \mathbf{x})$ is equal to $\frac{1}{Z}\exp(-E(\mathbf{x})/T)$ from Eq. (11.40).
- By definition, the state vector $\mathbf{x}$ is the joint combination of $\mathbf{x}_\alpha$ pertaining to the visible neurons and $\mathbf{x}_\beta$ pertaining to the hidden neurons. Hence, the probability of finding the visible neurons in state $\mathbf{x}_\alpha$ with any $\mathbf{x}_\beta$ is given by

$$P(\mathbf{X}_\alpha = \mathbf{x}_\alpha) = \frac{1}{Z}\sum_{\mathbf{x}_\beta} \exp\left(-\frac{E(\mathbf{x})}{T}\right) \tag{11.46}$$

where the random vector $\mathbf{X}_\alpha$ is a subset of $\mathbf{X}$. The partition function Z is itself defined by (see Eq. (11.6)):

$$Z = \sum_{\mathbf{x}} \exp\left(-\frac{E(\mathbf{x})}{T}\right) \tag{11.47}$$

Thus, substituting Eqs. (11.46) and (11.47) in (11.45), we obtain the desired expression for the log-likelihood function:

$$L(\mathbf{w}) = \sum_{\mathbf{x}_\alpha \in \mathcal{T}} \left(\log \sum_{\mathbf{x}_\beta} \exp\left(-\frac{E(\mathbf{x})}{T}\right) - \log \sum_{\mathbf{x}} \exp\left(-\frac{E(\mathbf{x})}{T}\right)\right) \tag{11.48}$$

The dependence on $\mathbf{w}$ is contained in the energy function $E(\mathbf{x})$, as shown in Eq. (11.39).

Differentiating $L(\mathbf{w})$ with respect to w_{ji} in light of Eq. (11.39), we obtain the following result after some manipulation of terms (see Problem 11.8):

$$\frac{\partial L(\mathbf{w})}{\partial w_{ji}} = \frac{1}{T}\sum_{\mathbf{x}_\alpha \in \mathcal{T}} \left(\sum_{\mathbf{x}_\beta} P(\mathbf{X}_\beta = \mathbf{x}_\beta | \mathbf{X}_\alpha = \mathbf{x}_\alpha) x_j x_i - \sum_{\mathbf{x}} P(\mathbf{X} = \mathbf{x}) x_j x_i\right) \tag{11.49}$$

To simplify matters, we introduce two definitions:

$$\begin{aligned} \rho_{ji}^{+} &= \langle x_j x_i \rangle^{+} \\ &= \sum_{\mathbf{x}_\alpha \in \mathcal{T}} \sum_{\mathbf{x}_\beta} P(\mathbf{X}_\beta = \mathbf{x}_\beta | \mathbf{X}_\alpha = \mathbf{x}_\alpha) x_j x_i \end{aligned} \tag{11.50}$$

and

$$\begin{aligned} \rho_{ji}^{-} &= \langle x_j x_i \rangle^{-} \\ &= \sum_{\mathbf{x}_\alpha \in \mathcal{T}} \sum_{\mathbf{x}} P(\mathbf{X} = \mathbf{x}) x_j x_i \end{aligned} \tag{11.51}$$

In a loose sense, we may view the first average, ρ_{ji}^{+}, as the mean firing rate or *correlation* between the states of neurons i and j with the network operating in its clamped or positive phase, and similarly view the second average, ρ_{ji}^{-}, as the *correlation* between

the states of neurons i and j with the network operating in its free-running or negative phase. With these definitions we may simplify Eq. (11.49) to

$$\frac{\partial L(\mathbf{w})}{\partial w_{ji}} = \frac{1}{T}\left(\rho_{ji}^{+} - \rho_{ji}^{-}\right) \tag{11.52}$$

The goal of Boltzmann learning is to maximize the log-likelihood function $L(\mathbf{w})$. We may use *gradient ascent* to achieve that goal by writing

$$\begin{aligned}\Delta w_{ji} &= \epsilon \frac{\partial L(\mathbf{w})}{\partial w_{ji}} \\ &= \eta\left(\rho_{ji}^{+} - \rho_{ji}^{-}\right)\end{aligned} \tag{11.53}$$

where η is a *learning-rate parameter*; it is defined in terms of ϵ and the operating temperature T as:

$$\eta = \frac{\epsilon}{T} \tag{11.54}$$

The gradient ascent rule of Eq. (11.53) is called the *Boltzmann learning rule*. The learning described here is performed in batch; that is, changes to the synaptic weights are made on the presentation of the entire set of training examples.

According to this learning rule, the synaptic weights of a Boltzmann machine are adjusted by using only locally available observations under two different conditions: (1) clamped, and (2) free running. This important feature of Boltzmann learning greatly simplifies the network architecture, particularly when dealing with large networks. Another useful feature of Boltzmann learning, which may come as a surprise, is that the rule for adjusting the synaptic weight from neuron i to neuron j is independent of whether these two neurons are both visible, both hidden, or one of each. All of these nice features of Boltzmann learning result from a key insight by Hinton and Sejnowski (1983, 1986), which ties the abstract mathematical model of the Boltzmann machine to neural networks by using a combination of two things:

- The Gibbs distribution for describing the stochasticity of a neuron.
- The statistical physics-based energy function of Eq. (11.39) for defining the Gibbs distribution.

From a learning point of view, the two terms that constitute the Boltzmann learning rule of Eq. (11.53) have opposite meaning. We may view the first term, corresponding to the clamped condition of the network, as essentially a Hebbian *learning* rule; and view the second term, corresponding to the free-running condition of the network, as an *unlearning* or *forgetting* term. Indeed, the Boltzmann learning rule represents a *generalization* of the *repeated forgetting and relearning rule* described by Pöppel and Krey (1987) for the case of symmetric networks with no hidden neurons.

It is also of interest that since the Boltzmann machine learning algorithm requires that hidden neurons know the difference between stimulated and free-running activities, and given that there is a (hidden) external network that signals to hidden neurons that the machine is being stimulated, we have a primitive form of an *attention* mechanism (Cowan and Sharp, 1988).

Need for the Negative Phase and its Implications

The combined use of a positive and a negative phase stabilizes the distribution of synaptic weights in the Boltzmann machine. This need may be justified in another way. Intuitively, we may say that the need for a negative as well as a positive phase in Boltzmann learning arises due to the presence of the partition function, Z, in the expression for the probability of a neuron's state vector. The implication of this statement is that the direction of steepest descent in energy space is *not* the same as the direction of steepest ascent in probability space. In effect, the negative phase in the learning procedure is needed to account for this discrepancy (Neal, 1992).

The use of a negative phase in Boltzmann learning has two major disadvantages:

1. *Increased computation time*. In the positive phase some of the neurons are clamped to the external environment, whereas in the negative phase all the neurons are free running. Accordingly, the time taken for stochastic simulation of a Boltzmann machine is increased.
2. *Sensitivity to statistical errors*. The Boltzmann learning rule involves the *difference* between two average correlations, one computed for the positive phase and the other computed for the negative phase. When these two correlations are similar, the presence of sampling noise makes the difference between them even more noisy.

We may eliminate these shortcomings of the Boltzmann machine by using a sigmoid belief network. In this new class of stochastic machines, control over the learning procedure is exercised by means other than a negative phase.

11.8 SIGMOID BELIEF NETWORKS

Sigmoid belief networks or *logistic belief nets* were developed by Neal in 1992 in an effort to find a stochastic machine that would share with the Boltzmann machine the capacity to learn arbitrary probability distributions over binary vectors, but would not need the negative phase of the Boltzmann machine learning procedure. This objective was achieved by replacing the symmetric connections of the Boltzmann machine with *directed connections that form an acyclic graph*. Specifically, a sigmoid belief network consists of a multilayer architecture with binary stochastic neurons, as illustrated in Fig. 11.6. The acyclic nature of the machine makes it easy to perform probabilistic calculations. In particular, the network uses the sigmoid function of Eq. (11.43), in analogy with the Boltzmann machine, to calculate the conditional probability of a neuron being activated in response to its own induced local field.

Fundamental Properties of Sigmoid Belief Networks

Let the vector $\mathbf{X}$, consisting of the two-valued random variables $X_1, X_2, \ldots, X_N$, define a sigmoid belief network composed of N stochastic neurons. The *parents* of element X_j in $\mathbf{X}$ are denoted by

$$\text{pa}(X_j) \subseteq \{X_1, X_2, \ldots, X_{j-1}\} \tag{11.55}$$

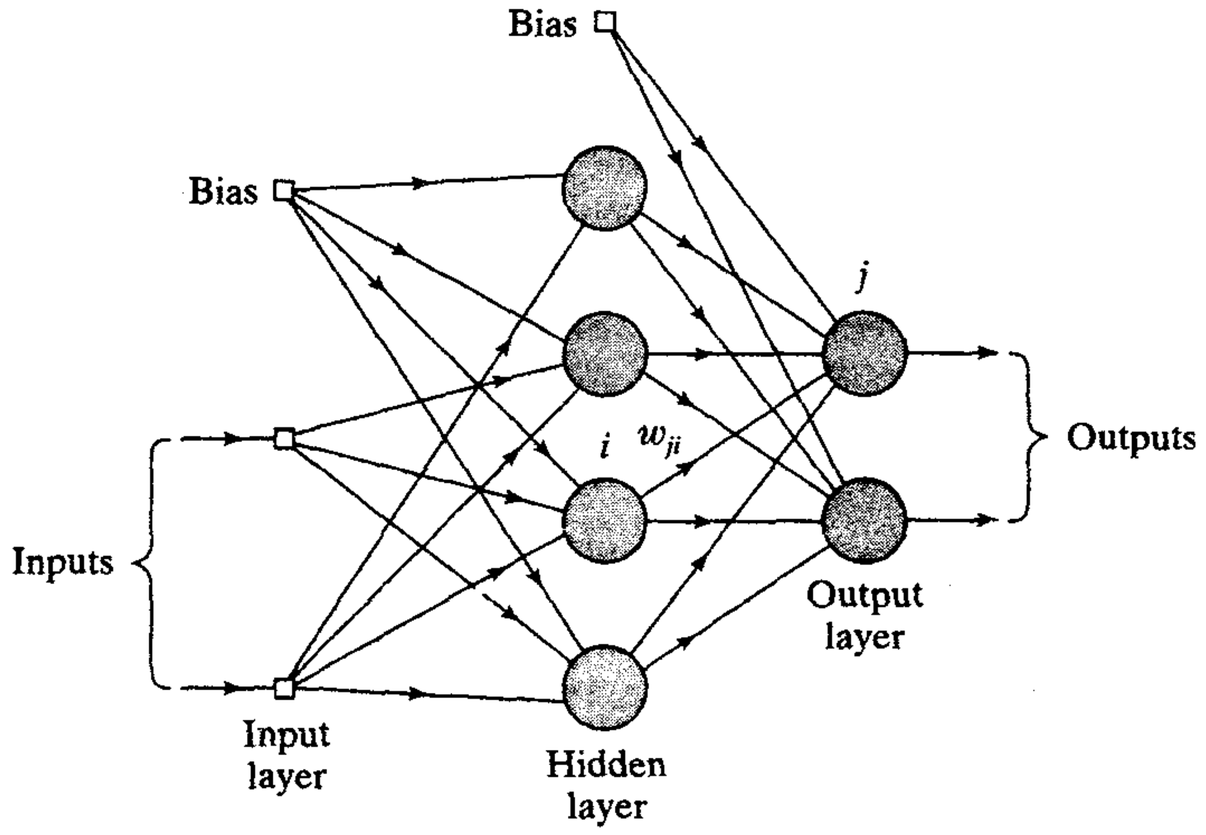


FIGURE 11.6 Architectural graph of sigmoid belief network.

In words, $\text{pa}(X_j)$ is the smallest subset of random vector $\mathbf{X}$ for which we have

$$P(X_j = x_j | X_1 = x_1, \ldots, X_{j-1} = x_{j-1}) = P(X_j = x_j | \text{pa}(X_j)) \tag{11.56}$$

An important virtue of sigmoid belief networks is their ability to clearly exhibit the conditional dependencies of the underlying probabilistic model of the input data. In particular, the probability that the ith neuron is activated is defined by the sigmoid function (see Eq. (11.43))

$$P(X_j = x_j | \text{pa}(X_j)) = \varphi\left(\frac{x_j}{T} \sum_{i<j} w_{ji} x_i\right) \tag{11.57}$$

where w_{ji} is the synaptic weight from neuron i to neuron j, as shown in Fig. 11.6. That is, the conditional probability $P(X_j = x_j | \text{pa}(X_j))$ depends on $\text{pa}(X_j)$ solely through a sum of weighted inputs. Thus, Eq. (11.57) provides the basis for the propagation of beliefs through the network.

In performing probability calculations on sigmoid belief networks, the following two points are noteworthy:

1. $w_{ji} = 0$ for all X_i not belonging to $\text{pa}(X_j)$
2. $w_{ji} = 0$ for all $i \geq j$

The first point follows from the definition of parents. The second point follows from the fact that a sigmoid belief network is a directed acyclic graph.

As the name implies, sigmoid belief networks belong to the general class of *belief networks*[9] studied extensively in the literature (Pearl, 1988). The stochastic operation of sigmoid belief networks is somewhat more complex than the Boltzmann machine. Nevertheless, they do lend themselves to the use of gradient-ascent learning in probability space, based on locally available information.

Learning in Sigmoid Belief Networks

Let $\mathcal{T}$ denote a set of training examples drawn from the probability distribution of interest. It is assumed that each example is two-valued, representing certain attributes. Repetition of training examples is permitted, in proportion to how commonly a particular combination of attributes is known to occur. To model the distribution from which $\mathcal{T}$ is drawn, we proceed as follows:

1. Some size for a state vector, $\mathbf{x}$, is decided for the network.
2. A subset of the state vector, say $\mathbf{x}_\alpha$, is selected to represent the attributes in the training cases; that is, $\mathbf{x}_\alpha$ represents the state vector of the visible neurons (i.e., evidence nodes).
3. The remaining part of the state vector $\mathbf{x}$, denoted by $\mathbf{x}_\beta$, defines the state vector of the hidden neurons (i.e., those computational nodes for which we do not have instantiated values).

The design of a sigmoid belief network is highly dependent on the way in which, for a given state vector $\mathbf{x}$, the visible and hidden units are arranged. Therefore, different arrangements of visible and hidden neurons may result in different configurations.

As with the Boltzmann machine, we derive the desired learning rule for a sigmoid belief network by maximizing the log-likelihood function, computed from the training set $\mathcal{T}$. The log-likelihood function, $L(\mathbf{w})$, is defined by Eq. (11.45), reproduced here for convenience of presentation:

$$L(\mathbf{w}) = \sum_{\mathbf{x}_\alpha \in \mathcal{T}} \log P(\mathbf{X}_\alpha = \mathbf{x}_\alpha)$$

where $\mathbf{w}$ is the synaptic weight vector of the network, treated as unknown. The state vector $\mathbf{x}_\alpha$, pertaining to the visible neurons, is a realization of the random vector $\mathbf{X}_\alpha$. Let w_{ji} denote the ji-th element of $\mathbf{w}$ (i.e., synaptic weight from neuron i to neuron j). Differentiating $L(\mathbf{w})$ with respect to w_{ji}, we obtain

$$\frac{\partial L(\mathbf{w})}{\partial w_{ji}} = \sum_{\mathbf{x}_\alpha \in \mathcal{T}} \frac{1}{P(\mathbf{X}_\alpha = \mathbf{x}_\alpha)} \frac{\partial P(\mathbf{X}_\alpha = \mathbf{x}_\alpha)}{\partial w_{ji}}$$

Next we note the following two probabilistic relations:

$$\begin{aligned} P(\mathbf{X}_\alpha = \mathbf{x}_\alpha) &= \sum_{\mathbf{x}_\beta} P(\mathbf{X} = (\mathbf{x}_\alpha, \mathbf{x}_\beta)) \\ &= \sum_{\mathbf{x}_\beta} P(\mathbf{X} = \mathbf{x}) \end{aligned} \tag{11.58}$$

where the random vector $\mathbf{X}$ pertains to the whole network and the state vector $\mathbf{x} = (\mathbf{x}_\alpha, \mathbf{x}_\beta)$ is a realization of it, and

$$P(\mathbf{X} = \mathbf{x}) = P(\mathbf{X} = \mathbf{x} | \mathbf{X}_\alpha = \mathbf{x}_\alpha) P(\mathbf{X}_\alpha = \mathbf{x}_\alpha) \tag{11.59}$$

which defines the probability of the joint event $\mathbf{X} = \mathbf{x} = (\mathbf{x}_\alpha, \mathbf{x}_\beta)$.

In light of these two relations, we may redefine the partial derivative $\partial L(\mathbf{w})/\partial w_{ji}$ in the equivalent form:

$$\frac{\partial L(\mathbf{w})}{\partial w_{ji}} = \sum_{\mathbf{x}_\alpha \in \mathcal{T}} \sum_{\mathbf{x}_\beta} \frac{P(\mathbf{X} = \mathbf{x}|\mathbf{X}_\alpha = \mathbf{x}_\alpha)}{P(\mathbf{X} = \mathbf{x})} \frac{\partial P(\mathbf{X} = \mathbf{x})}{\partial w_{ji}} \tag{11.60}$$

In light of Eq. (11.43), we may write

$$P(\mathbf{X} = \mathbf{x}) = \prod_j \varphi\left(\frac{x_j}{T} \sum_{i<j} w_{ji} x_i\right) \tag{11.61}$$

where $\varphi(\cdot)$ is a sigmoid function. We may therefore write

$$\begin{aligned}
\frac{1}{P(\mathbf{X} = \mathbf{x})} \frac{\partial P(\mathbf{X} = \mathbf{x})}{\partial w_{ji}} &= \frac{\partial}{\partial w_{ji}} \log P(\mathbf{X} = \mathbf{x}) \\
&= \frac{\partial}{\partial w_{ji}} \sum_j \log \varphi\left(\frac{x_j}{T} \sum_{i<j} w_{ji} x_i\right) \\
&= \frac{1}{T} \sum_j \frac{1}{\varphi\left(\frac{x_j}{T} \sum_{i<j} w_{ji} x_i\right)} \varphi'\left(\frac{x_j}{T} \sum_{i<j} w_{ji} x_i\right) x_j x_i
\end{aligned}$$

where $\varphi'(\cdot)$ is the first derivative of the sigmoid function $\varphi(\cdot)$ with respect to its argument. But, from the definition of $\varphi(\cdot)$ given in Eq. (11.44) we readily find that

$$\varphi'(v) = \varphi(v)\varphi(-v) \tag{11.62}$$

where $\varphi(-v)$ is obtained from $\varphi(v)$ by replacing v with $-v$. Hence, we may write

$$\frac{1}{P(\mathbf{X} = \mathbf{x})} \frac{\partial P(\mathbf{X} = \mathbf{x})}{\partial w_{ji}} = \frac{1}{T} \sum_j \varphi\left(-\frac{x_j}{T} \sum_{i<j} w_{ji} x_i\right) x_j x_i \tag{11.63}$$

Accordingly, substituting Eq. (11.63) in (11.60), we obtain

$$\frac{\partial L(\mathbf{w})}{\partial w_{ji}} = \frac{1}{T} \sum_{\mathbf{x}_\alpha \in \mathcal{T}} \sum_{\mathbf{x}_\beta} P(\mathbf{X} = \mathbf{x}|\mathbf{X}_\alpha = \mathbf{x}_\alpha) \varphi\left(-\frac{x_j}{T} \sum_{i<j} w_{ji} x_i\right) x_j x_i \tag{11.64}$$

To simplify matters, we define the ensemble average

$$\begin{aligned}
\rho_{ji} &= \left\langle \varphi\left(-x_j \sum_{i<j} w_{ji} x_i\right) x_j x_i \right\rangle \\
&= \sum_{\mathbf{x}_\alpha \in \mathcal{T}} \sum_{\mathbf{x}_\beta} P(\mathbf{X} = \mathbf{x}|\mathbf{X}_\alpha = \mathbf{x}_\alpha) \varphi\left(-\frac{x_j}{T} \sum_{i<j} w_{ji} x_i\right) x_j x_i
\end{aligned} \tag{11.65}$$

which represents an *average correlation* between the states of neurons i and j, weighted by the factor $\varphi(-\frac{x_j}{T} \sum_{i<j} w_{ji} x_i)$. This average is taken over all possible values of $\mathbf{x}_\alpha$ (drawn from the training set $\mathcal{T}$) as well as all possible values of $\mathbf{x}_\beta$, with $\mathbf{x}_\alpha$ referring to the visible neurons and $\mathbf{x}_\beta$ referring to the hidden neurons.

Gradient ascent in probability space is accomplished by defining the incremental change in synaptic weight w_{ji} as

$$\begin{aligned}\Delta w_{ji} &= \epsilon \frac{\partial L(\mathbf{w})}{\partial w_{ji}} \\ &= \eta \rho_{ji}\end{aligned} \tag{11.66}$$

where $\eta = \epsilon/T$ is a learning-rate parameter and ρ_{ji} is itself defined by Eq. (11.65). Equation (11.66) is the *learning rule for a sigmoid belief network*.

A summary of the sigmoid belief network learning procedure is presented in Table 11.2, where learning is performed in the batch mode; that is, the changes to the synaptic weights of the network are made on the basis of the entire set of training cases. The summary presented in Table 11.2 does not include the use of simulated annealing, which is why we have set the temperature T to 1 therein. However, as with the Boltzmann machine, simulated annealing can be incorporated into the sigmoid belief network learning procedure to reach thermal equilibrium faster, if so desired.

TABLE 11.2 Summary of the Sigmoid Belief Network Learning Procedure

Initialization. Initialize the network by setting the weights w_{ji} of the network to random values uniformly distributed in the range $[-a, a]$; a typical value for a is 0.5.

1. Given a set of training cases $\mathcal{T}$, clamp the visible neurons of the network to $\mathbf{x}_\alpha$, where $\mathbf{x}_\alpha \in \mathcal{T}$.
2. For each $\mathbf{x}_\alpha$, perform a separate Gibbs sampling simulation of the network at some operating temperature T, and observe the resulting state vector $\mathbf{x}$ of the whole network. Provided that the simulation is performed long enough, the values of $\mathbf{x}$ for the different cases contained in the training set $\mathcal{T}$ should come from the conditional distribution of the corresponding random vector $\mathbf{X}$, given that particular training set.
3. Compute the ensemble average

$$\rho_{ji} = \sum_{\mathbf{x}_\alpha \in \mathcal{T}} \sum_{\mathbf{x}_\beta} P(\mathbf{X} = \mathbf{x} | \mathbf{X}_\alpha = \mathbf{x}_\alpha) x_j x_i \varphi\left(-x_j \sum_{i<j} w_{ji} x_i\right)$$

where the random vector $\mathbf{X}_\alpha$ is a subset of $\mathbf{X}$, and $\mathbf{x} = (\mathbf{x}_\alpha, \mathbf{x}_\beta)$ with $\mathbf{x}_\alpha$ referring to the visible neurons and $\mathbf{x}_\beta$ referring to the hidden neurons; x_j is the jth element of state vector $\mathbf{x}$ (i.e., state of neuron j), and w_{ji} is the synaptic weight from neuron i to neuron j. The sigmoid function $\varphi(\cdot)$ is defined by

$$\varphi(v) = \frac{1}{1 + \exp(-v)}$$

4. Increment each synaptic weight w_{ji} of the network by the amount

$$\Delta w_{ji} = \eta \rho_{ji}$$

where η is the learning-rate parameter. This adjustment should move the synaptic weights of the network along the gradient toward a local maximum of the log-likelihood function $L(\mathbf{w})$ in accordance with the maximum likelihood principle.

Unlike the Boltzmann machine, only a single phase is needed for learning in a sigmoid belief network. The reason for this simplification is that normalization of the probability distribution over state vectors is accomplished at the local level of each neuron via the sigmoid function $\varphi(\cdot)$, rather than globally via the difficult to compute partition function Z that involves all possible configurations of states. Once the conditional distribution of the random vector $\mathbf{X}$, given the values of $\mathbf{x}_\alpha$ drawn from the training set $\mathcal{T}$, has been correctly modeled via Gibbs sampling, the role of the negative phase in the Boltzmann machine learning procedure is taken over by the weighting factor $\varphi(-\frac{x_j}{T}\sum_{i<j} w_{ji}x_i)$ involved in computing the ensemble-averaged correlation ρ_{ji} between the states of neurons i and j. When the local minimum of the log-likelihood function $L(\mathbf{w})$ is reached, this weighting factor becomes zero when the network learns a deterministic mapping; otherwise, its average effect comes out to zero.

In Neal (1992), experimental results are presented that show (1) sigmoid belief networks are capable of learning to model nontrivial distributions, (2) these networks can learn at a faster rate than the Boltzmann machine, and (3) this advantage of a sigmoid belief network over the Boltzmann machine is due to the elimination of the negative phase from the learning procedure.

11.9 HELMHOLTZ MACHINE

Sigmoid belief networks provide a powerful multilayer framework for representing and learning higher-order statistical relationships among sensory inputs of interest in an unsupervised manner. The *Helmholtz machine,*[10] first described in Dayan et al. (1995) and Hinton et al. (1995), provides another ingenious multilayer framework for achieving a similar objective without the use of Gibbs sampling.

The Helmholtz machine uses two entirely different sets of synaptic connections, as illustrated in Fig. 11.7 for the case of a network with two layers of two-valued, stochastic neurons. The forward connections, shown as solid lines in Fig. 11.7, constitute

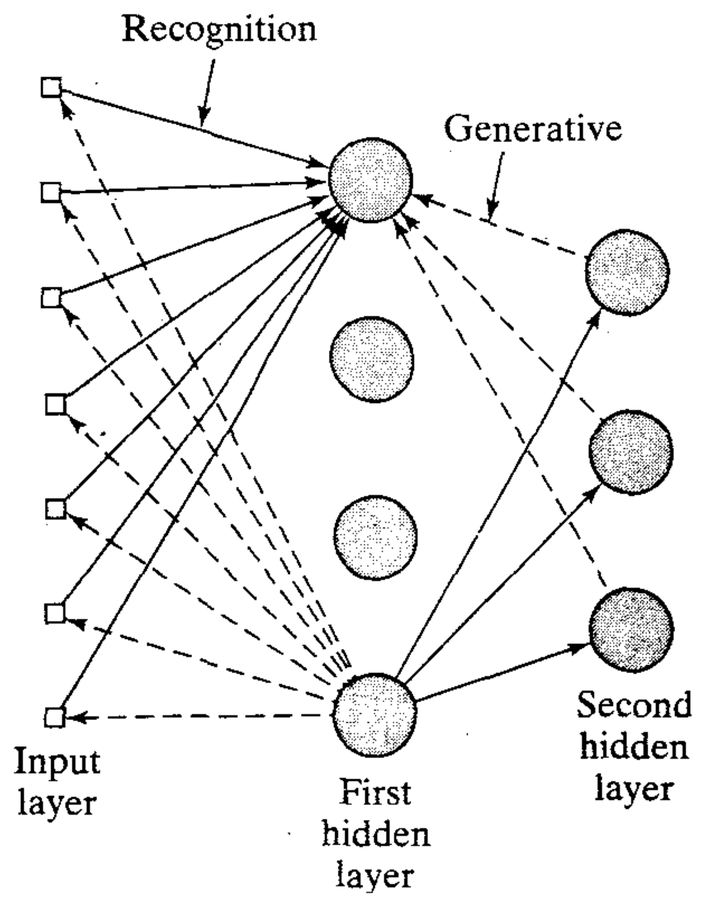


FIGURE 11.7 Architectural graph of Helmholtz machine consisting of interconnected neurons with recognition (solid lines) and generative (dashed lines) connections.

the *recognition model*. The purpose of this model is to *infer* a probability distribution over the underlying causes of the input vector. The backward connections, shown as dashed lines in Fig. 11.7, constitute the *generative model*. The purpose of this second model is to *reconstruct* an approximation to the original input vector from the underlying representations captured by the hidden layers of the network, thereby enabling it to operate in a *self-supervised manner*. Both the recognition and generative models operate in a strictly feedforward fashion, with no feedback; they interact with each other only via the learning procedure.

Hinton et al. (1995) describe a stochastic algorithm, called the *wake-sleep algorithm*, for calculating the recognition and generative weights of the Helmholtz machine. As the name implies, there are two phases to the algorithm: a "wake" phase and a "sleep" phase. In the "wake" phase, the network is driven in the forward direction by the recognition weights. A representation of the input vector is thereby produced in the first hidden layer. This is, in turn, followed by a second representation of that first representation, which is produced in the second hidden layer, and so on for the other hidden layers of the network. The set of representations so produced in the different hidden layers of the network provides a total representation of the input vector by the network. Although the neurons are driven by the recognition weights, it is only the generative weights that are actually learned during the "wake" phase using locally available information. In effect, this phase of the learning process makes each layer of the total representation better at reconstructing the activities formed in the preceding layer.

In the "sleep" phase of the algorithm, the recognition weights are turned off. The network is driven in the backward direction by the generative weights, starting at the outermost hidden layer and working backwards, layer by layer, all the way to the input layer. Because of the fact that the neurons are stochastic, repeating this process would typically give rise to many different "fantasy" vectors on the input layer. These fantasies supply an unbiased sample of the network's generative model of the world. Having produced a fantasy, the simple delta rule (described in Chapter 3) is used to adjust the recognition weights so as to maximize the logarithm of the probability of recovering the hidden activities that actually caused the fantasy. Like the "wake" phase, the "sleep" phase only uses locally available information.

The learning rule for the generative weights (i.e., backward connections) also uses the simple delta rule. However, instead of following the gradient of the log-likelihood function, this rule follows the gradient of a *penalized* log-likelihood function. The penalty term is the Kullback–Leibler divergence between the true *a posteriori* distribution and the actual distribution produced by the recognition model (Hinton et al., 1995); the Kullback–Leibler divergence or relative entropy is discussed in the preceding chapter. In effect, the penalized log-likelihood function acts as a lower bound on the log-likelihood function of the input data, with the lower bound being improved through the learning process. In particular, the learning process tries to adjust the generative weights to bring the true *a posteriori* distribution as close as possible to the distribution actually computed by the recognition model. Unfortunately, learning the recognition model's weights does not precisely correspond to the penalized likelihood function. The wake-sleep learning procedure is not guaranteed to work in all practical situations; it does fail sometimes.

11.10 MEAN-FIELD THEORY

The learning machines considered in the preceding three sections share a common feature: they all use stochastic neurons and may therefore suffer from a slow learning process. In the third and final part of the chapter we study the use of mean-field theory as the mathematical basis for deriving *deterministic approximations* to these stochastic machines to speed up learning. Because the stochastic machines considered herein have different architectures, the theory is applied in correspondingly different ways. In particular, we may identify two specific approaches that have been pursued in literature:

1. Correlations are replaced by their mean-field approximations.
2. An intractable model is replaced by a tractable model via a variational principle.

Approach 2 is highly principled and therefore very appealing. It lends itself for application to the sigmoid belief network (Saul et al., 1996) and the Helmholtz machine (Dayan et al., 1995). However, in the case of the Boltzmann machine, the application of approach 2 is complicated by the need for an upper bound on the partition function Z. For this reason, in Peterson and Anderson (1987) , the first approach is used to accelerate the Boltzmann learning rule. In this section we provide a rationale for the first approach. The second approach is considered later in the chapter.

The idea of mean-field approximation is well known in statistical physics (Glauber, 1963). While it cannot be denied that in the context of stochastic machines it is desirable to know the states of all the neurons in the network at all times, we must nevertheless recognize that, in the case of a network with a large number of neurons, the neural states contain vastly more information than we usually require in practice. In fact, to answer the most familiar physical questions about the stochastic behavior of the network, we need only know the average values of neural states or the average products of pairs of neural states.

In a stochastic neuron, the firing mechanism is described by a probabilistic rule. In such a situation, it is rational for us to enquire about the *average* of the state x_j of neuron j. To be precise, we should speak of the average as a "thermal" average, since the synaptic noise is usually modeled in terms of thermal fluctuations. In any event, let $<x_j>$ denote the average (mean) of x_j. The state of neuron j is described by the probabilistic rule:

$$x_j = \begin{cases} +1 & \text{with probability } P(v_j) \\ -1 & \text{with probability } 1 - P(v_j) \end{cases} \tag{11.67}$$

where

$$P(v_j) = \frac{1}{1 + \exp(-v_j/T)} \tag{11.68}$$

where T is the operating temperature. Hence we may express the average $<x_j>$ for some *specified* value of induced local field v_j as follows:

$$\begin{aligned} <x_j> &= (+1)P(v_j) + (-1)[1 - P(v_j)] \\ &= 2P(v_j) - 1 \\ &= \tanh(v_j/2T) \end{aligned} \tag{11.69}$$

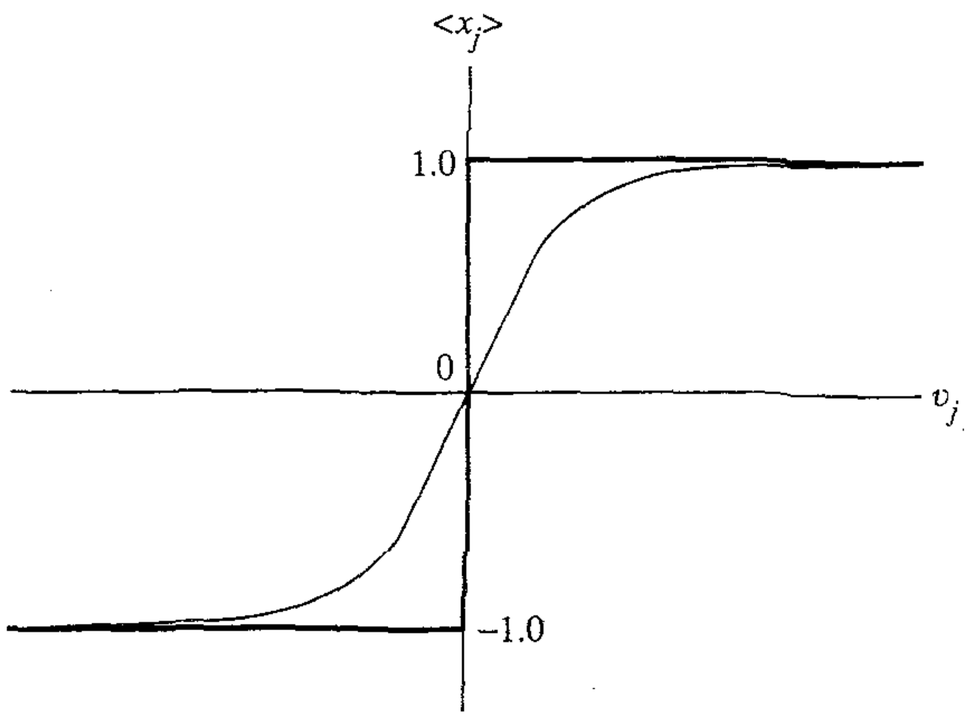


FIGURE 11.8 Graph of the thermal average $<x_j>$ versus the induced local field v_j; heavy solid curve corresponds to normal operation of the McCulloch-Pitts neuron.

where $\tanh(v_j/2T)$ is the hyperbolic tangent of $v_j/2T$. Figure 11.8 shows two plots of the average $<x_j>$ versus the induced local field v_j. The continuous curve is for some temperature T greater than zero, and the plot shown in heavy solid lines is for the limiting case of $T = 0$. In the latter case, Eq. (11.69) takes the limiting form

$$<x_j> \rightarrow \operatorname{sgn}(v_j) \quad \text{as } T \rightarrow 0 \tag{11.70}$$

which corresponds to the activation function of the McCulloch–Pitts neuron.

The discussion so far has focused on the simple case of a single stochastic neuron. In the more general case of a stochastic machine composed of a large assembly of neurons, we have a much more difficult task on our hands. The difficulty arises because of the combination of two factors:

- The probability $P(v_j)$ that neuron j is on is a nonlinear function of the induced local field v_j.
- The induced local field v_j is a random variable, being influenced by the stochastic action of other neurons connected to the inputs of neuron j.

It is generally safe to say that there is no mathematical method that we can use to evaluate the behavior of a stochastic machine in exact terms. But there is an approximation known as the *mean-field approximation* that we can use, which often yields good results. The basic idea of mean-field approximation is to replace the actual fluctuating induced local field v_j for each neuron j in the network by its average $<v_j>$, as shown by

$$v_j \overset{\text{approx}}{=} <v_j> = \left\langle \sum_i w_{ji} x_i \right\rangle = \sum_i w_{ji} <x_i> \tag{11.71}$$

Accordingly, we may compute the average state $<x_j>$ for neuron j embedded in a stochastic machine made up of a total of N neurons, just as we did in Eq. (11.69) for a single stochastic neuron, by writing

$$<x_j> = \tanh\left(\frac{1}{2T}v_j\right) \overset{\text{approx}}{=} \tanh\left(\frac{1}{2T}<v_j>\right) = \tanh\left(\frac{1}{2T}\sum_i w_{ji}<x_j>\right) \quad (11.72)$$

In light of Eq. (11.72), we may formally state the mean-field approximation as:

The average of some function of a random variable is approximated as the function of the average of that random variable.

For $j = 1, 2, \ldots, N$, Eq. (11.72) represents a set of nonlinear equations with N unknowns $<x_j>$. The solution of this set of nonlinear equations is now a manageable proposition because the unknowns are all *deterministic* rather than stochastic variables, which they are in the original network.

11.11 DETERMINISTIC BOLTZMANN MACHINE

Learning in the Boltzmann machine is *exponential* in the number of neurons because the Boltzmann learning rule requires the computation of correlations between every pair of neurons in the network. Boltzmann learning therefore requires exponential time. Peterson and Anderson (1987) proposed a method for accelerating the Boltzmann learning process. The method involves replacing the correlations in the Boltzmann learning rule of Eq. (11.53) by a mean-field approximation as shown here:

$$<x_j x_i> \overset{\text{approx}}{=} <x_j><x_i>, \quad (i, j) = 1, 2, \ldots, K \quad (11.73)$$

where the mean quantity $<x_j>$ is itself computed using the mean-field equation (11.72).

The form of Boltzmann learning in which the computation of the correlations is approximated in the manner just described is called the *deterministic Boltzmann learning rule*. Specifically, the standard Boltzmann learning rule of Eq. (11.53) is approximated as:

$$\Delta w_{ji} = \eta(U_j^+ U_i^+ - U_j^- U_i^-) \quad (11.74)$$

where U_j^+ and U_j^- are the average outputs of visible neuron j (on a single pattern) in the clamped and free-running conditions, respectively, and η is the learning-rate parameter. Whereas the Boltzmann machine uses binary stochastic neurons, its deterministic counterpart uses analog deterministic neurons.

The deterministic Boltzmann machine provides a substantial increase in speed over the standard Boltzmann machine by one or two orders of magnitude (Peterson and Anderson, 1987). However, there are some cautionary notes on its practical use:

1. The deterministic Boltzmann learning rule only works in the supervised case, that is, when some of the visible neurons are assigned the role of output neurons. Unsupervised learning does not work at all in the mean-field regime because the mean state is a very impoverished representation of the free-running probability distribution.