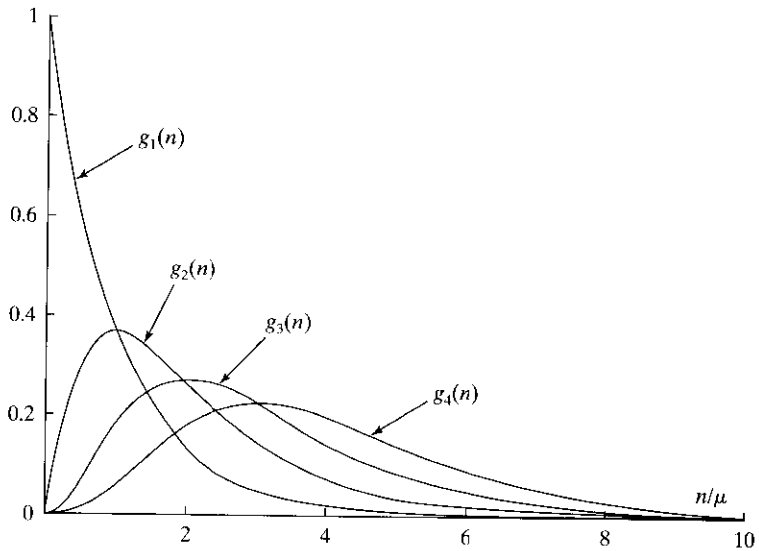


FIGURE 13.5 Family of impulse responses for the gamma memory for order $p = 1, 2, 3, 4$ and $\mu = 0.7$.

name of the memory. Figure 13.5 shows a family of impulse responses $g_p(n)$, normalized with respect to μ, for $p = 1, 2, 3, 4$ and $\mu = 0.7$. Note that the time axis in Fig. 13.5 is scaled by the parameter μ. This scaling has the effect of positioning the peak value of $g_p(n)$ at $n = p$.

The depth of the gamma memory is p/μ and its resolution is μ, for a depth-resolution product of p. Accordingly, by choosing μ to be less than unity, the gamma memory provides improvement in depth (but sacrifices resolution) over the tapped delay line for a specified order p. When $\mu = 1$, these quantities reduce to the respective values assumed by the tapped delay line. Thus the gamma memory includes the tapped delay line as a special case. This observation is also readily ascertained by setting $\mu = 1$ in Eq. (13.9). If μ is greater than 1 but less than 2, then $(1 - \mu)$ in this equation becomes negative but with an absolute value less than 1.

13.3 NETWORK ARCHITECTURES FOR TEMPORAL PROCESSING

Network architectures for temporal processing take more than one form, just as memory structures do. In this section we will describe two feedforward network architectures that have enriched the literature on temporal processing in their own individual ways.

NETtalk

NETtalk, devised by Sejnowski and Rosenberg (1987), was the first demonstration of a massively parallel distributed network that converts English speech to phonemes; a *phoneme* is a basic linguistic unit. Figure 13.6 shows a schematic diagram of the NETtalk system, which is based on a multilayer perceptron with an input layer of 203 sensory nodes, a hidden layer of 80 neurons, and an output layer of 26 neurons. All the

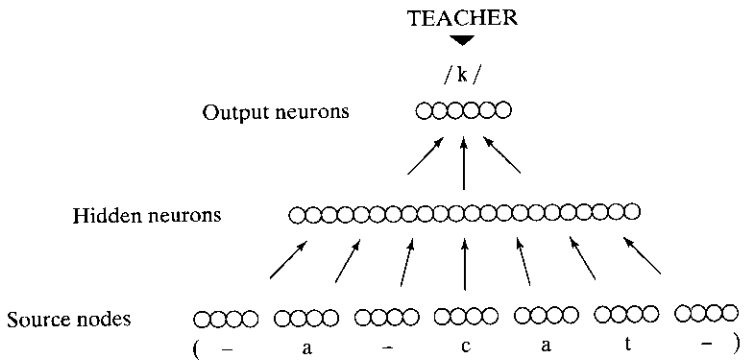


FIGURE 13.6 Schematic diagram of the NETtalk network architecture.

neurons used sigmoid (logistic) activation functions. The synaptic connections in the network were specified by a total of 18,629 weights, including a variable threshold for each neuron; threshold is the negative of bias. The standard back-propagation algorithm was used to train the network.

The network had seven groups of nodes in the input layer, with each group encoding one letter of the input text. Strings of seven letters were thus presented to the input layer at any one time. The desired response for the training process was specified as the correct phoneme associated with the center (i.e., fourth) letter in the seven-letter window. The other six letters (three on either side of the center letter) provided a partial *context* for each decision made by the network. The text was stepped through the window on a letter-by-letter basis. At each step in the process, the network computed a phoneme, and after each word the synaptic weights of the network were adjusted according to how closely the computed pronunciation matched the correct one.

The performance of NETtalk exhibited some similarities with observed human performance, as summarized here (Sejnowski and Rosenberg, 1987).

- The training followed a power law.
- The more words the network learned, the better it was at generalizing and correctly pronouncing new words.
- The performance of the network degraded very slowly as synaptic connections in the network were damaged.
- Relearning after damage to the network was much faster than learning during the original training.

NETtalk was a brilliant illustration in miniature of many aspects of learning, starting out with considerable "innate" knowledge of its input patterns and then gradually acquiring competence at converting English speech to phonemes through practice. However, it did not lead to practical applications.

Time-delay Neural Network

A popular neural network that uses ordinary time delays to perform temporal processing is the so-called *time delay neural network* (TDNN), which was first described in Lang and Hinton (1988) and Waibel et al. (1989). The TDNN is a multilayer feedforward

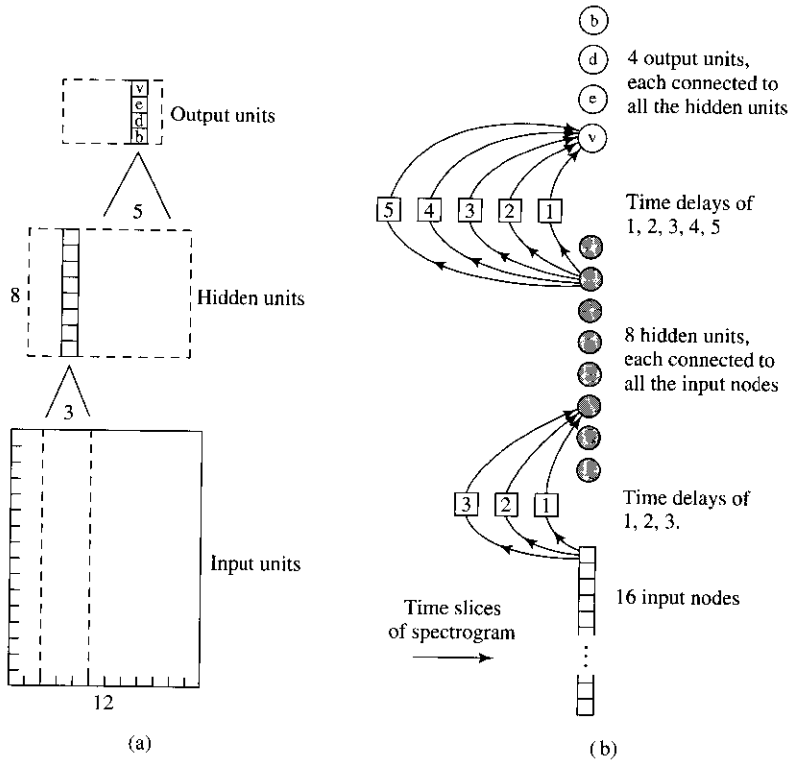


FIGURE 13.7 (a) A network whose hidden neurons and output neurons are replicated across time. (b) Time delay neural network (TDNN) representation. (From K. J. Lang and G. E. Hinton, 1988, with permission)

network whose hidden neurons and output neurons are *replicated across time*. It was devised to capture explicitly the concept of time symmetry as encountered in the recognition of an isolated word (phoneme) using a spectrogram. A *spectrogram* is a two-dimensional image in which the vertical dimension corresponds to frequency and the horizontal dimension corresponds to time; the intensity (darkness) of the image corresponds to signal energy (Rabiner and Schafer, 1978). Figure 13.7a illustrates a single hidden layer version of the TDNN (Lang and Hinton, 1988). The input layer consists of 192 (16 by 12) sensory nodes encoding the spectrogram; the hidden layer contains 10 copies of 8 hidden neurons; and the output layer contains 6 copies of 4 output neurons. The various replicas of a hidden neuron apply the same set of synaptic weights to narrow (three-time-step) windows of the spectrogram; similarly, the various replicas of an output neuron apply the same set of synaptic weights to narrow (five-time-step) windows of the pseudospectrogram computed by the hidden layer. Figure 13.7b presents a *time delay* interpretation of the replicated neural network of Fig. 13.7a—hence the name "time delay neural network". This network has a total of 544 synaptic weights. Lang and Hinton (1988) used the TDNN for the recognition of four isolated words: "bee", "dee", "ee", and "vee", which accounts for the use of four output neurons

in Fig. 13.7. A recognition score of 93 percent was obtained on test data different from the training data. In a more elaborate study reported by Waibel et al. (1989), a TDNN with two hidden layers was used for the recognition of three isolated words: "bee", "dee", and "gee". In performance evaluation involving the use of test data from three speakers, the TDNN achieved an average recognition score of 98.5 percent.

The TDNN appears to work best for classifying a temporal pattern that consists of a sequence of fixed dimensional feature vectors such as phonemes. In a practical speech recognizer, however, it is unrealistic to assume that the speech signal can be accurately segmented into its constituent phonemes. Rather, it is essential to adequately model the super-segmented temporal structure of speech patterns. In particular, the speech recognizer has to deal with word and sentence segments that vary significantly in their duration and nonlinear temporal structure. To model these natural characteristics of speech signals, the traditional approach in the speech recognition field has been to use a state transition structure like the hidden Markov model (Rabiner, 1989; Jelinek, 1997). Basically, a *hidden Markov model* (HMM) represents a stochastic process generated by an underlying Markov chain, and a set of observation distributions associated with its hidden states; see note 11 in Chapter 11. Many hybrids of TDNN and HMM have been studied in the literature.[4]

13.4 FOCUSED TIME LAGGED FEEDFORWARD NETWORKS

The prototypical use of a static neural network (e.g., multilayer perceptron, radial-basis function network) is in *structural pattern recognition*. In contrast, *temporal pattern recognition* requires processing of patterns that evolve over time, with the response at a particular instant of time depending not only on the present value of the input but also on its past values. Figure 13.8 shows the block diagram of a *nonlinear filter* built on a static neural network (Mozer, 1994). The network is stimulated through a short-term memory. Specifically, given an input signal consisting of the present value $x(n)$ and the p past values $x(n-1), \ldots, x(n-p)$ stored in a delay line memory of order p, for example, the free parameters of the neural network are adjusted to minimize the mean-square error between the output of the network, $y(n)$, and the desired response $d(n)$.

The structure of Fig. 13.8 can be implemented at the level of a single neuron or a network of neurons. These two cases are illustrated in Figs. 13.9 and 13.10 respectively.

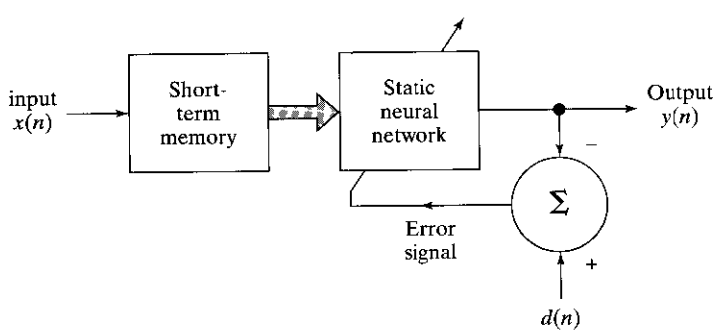


FIGURE 13.8 Nonlinear filter built on a static neural network.

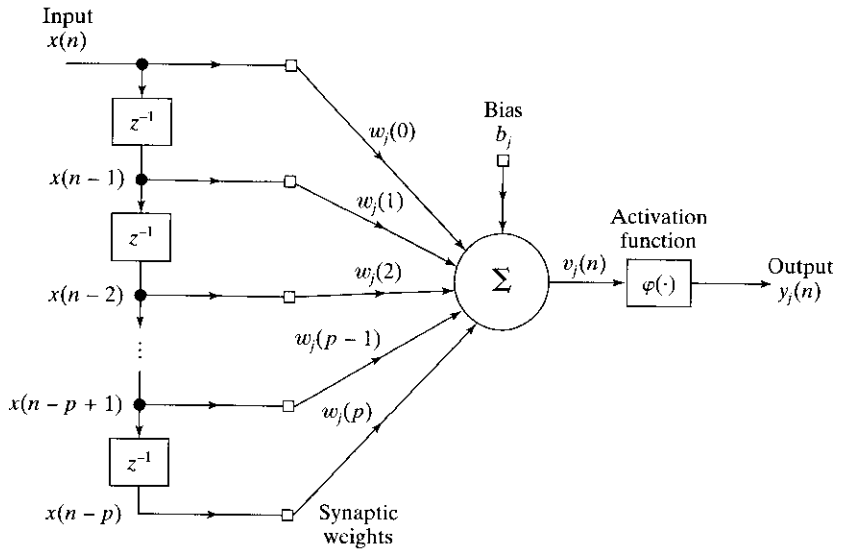


FIGURE 13.9 Focused neuronal filter.

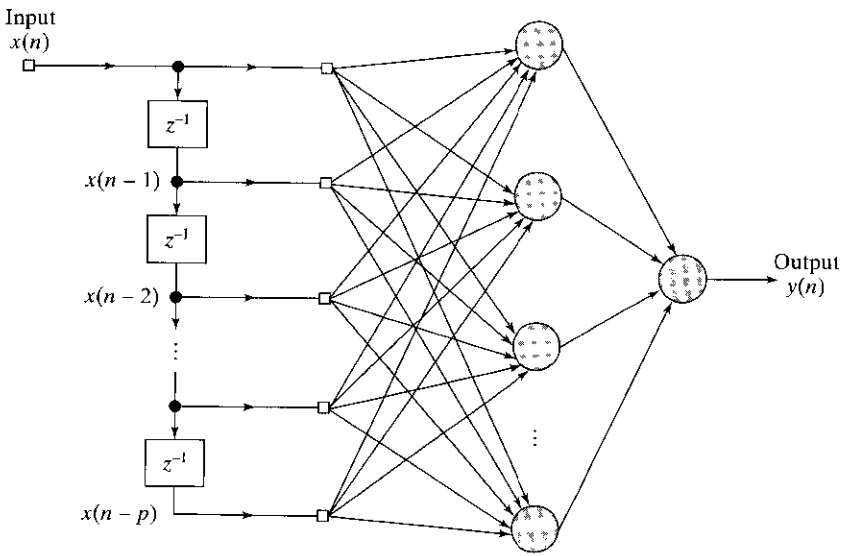


FIGURE 13.10 Focused time lagged feedforward network (TLFN); the bias levels have been omitted for convenience of presentation.

To simplify the presentation, we have used a tapped delay line memory as the short-term memory structure in Figs. 13.9 and 13.10. Clearly, both of these figures could be generalized by using a unit with transfer function $G(z)$ in place of z^{-1}.

The temporal processing unit of Fig. 13.9 is composed of a tapped delay line memory with its taps connected to the synapses of a neuron. The tapped delay line memory captures temporal information contained in the input signal and the neuron embeds that information in its own synaptic weights. The processing unit of Fig. 13.9 is

called a *focused neuronal filter*, focused in the sense that the entire memory structure is located at the input end of the unit. The output of the filter, in response to the input $x(n)$ and its past values $x(n-1), \ldots, x(n-p)$, is given by

$$y_j(n) = \varphi\left(\sum_{l=0}^{p} w_j(l)x(n-l) + b_j\right) \tag{13.11}$$

where $\varphi(\cdot)$ is the activation function of neuron j, the $w_j(l)$ are its synaptic weights, and b_j is the bias. Note that the input to the activation function consists of a bias plus the *convolution* of sequences of input samples and synaptic weights of the neuron.

Turning next to Fig. 13.10, referred to as a *focused time lagged feedforward network* (TLFN), here we have a more powerful nonlinear filter consisting of a tapped delay line memory of order p and a multilayer perceptron. To train the filter, we may use the standard back-propagation algorithm described in Chapter 4. At time n, the "temporal pattern" applied to the input layer of the network is the signal vector

$$x(n) = [x(n), x(n-1), \ldots, x(n-p)]^T$$

which may be viewed as a description of the *state* of the nonlinear filter at time n. An epoch consists of a sequence of states (patterns), the number of which is determined by the memory order p and the size N of the training sample.

The output of the nonlinear filter, assuming that the multilayer perceptron has a single hidden layer as shown in Fig. 13.10, is given by

$$\begin{aligned} y(n) &= \sum_{j=1}^{m_1} w_j y_j(n) \\ &= \sum_{j=1}^{m_1} w_j \varphi\left(\sum_{l=0}^{p} w_j(l)x(n-l) + b_j\right) + b_o \end{aligned} \tag{13.12}$$

where the output neuron in the focused TLFN is assumed to be linear; the synaptic weights of the output neuron are denoted by the set $\{w_j\}_{j=1}^{m_1}$, where m_1 is the size of the hidden layer, and the bias is denoted by b_o.

13.5 COMPUTER EXPERIMENT

In this computer experiment, we investigate the use of the focused TLFN in Fig. 13.10 to simulate a time series representing a difficult frequency modulated signal:

$$x(n) = \sin(n + \sin(n^2)), \qquad n = 0, 1, 2, \ldots$$

The network was used as a *one-step predictor* with $x(n+1)$ providing the desired response for an input consisting of the set $\{x(n-l)\}_{l=0}^{p}$. The composition of the network and its parameters were as follows:

Order of tapped delay line memory, p:	20
Hidden layer, m_1:	10 neurons
Activation function of hidden neurons:	logistic
Output layer:	1 neuron
Activation function of output neuron:	linear
Learning-rate parameter (both layers):	0.01
Momentum constant:	none

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.

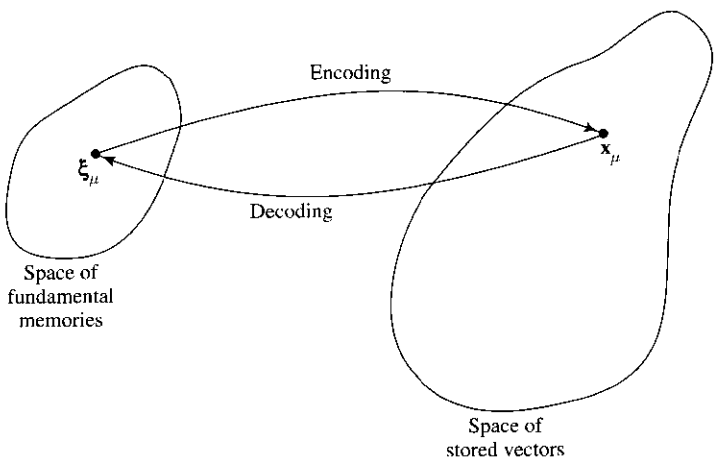


FIGURE 14.13 Illustration of the encoding–decoding performed by a recurrent network.

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

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

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 ξ_μ equal ± 1. Once they are computed, the synaptic weights are kept fixed.
2. *Initialization.* Let ξ_{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 ξ_{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.

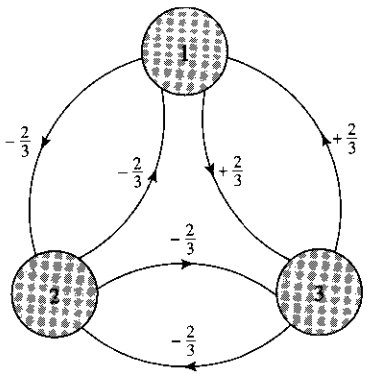
Table 14.2 presents a summary of the steps involved in the storage phase and retrieval phase of operating a Hopfield network.

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)

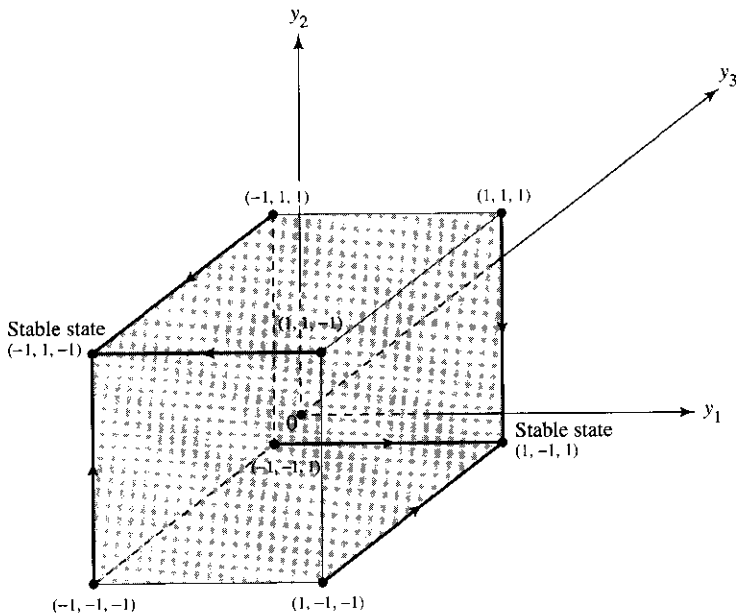


FIGURE 14.14
(a) Architectural graph of Hopfield network for $N = 3$ neurons. (b) Diagram depicting the two stable states and flow of the network.

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

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 ξ_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

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

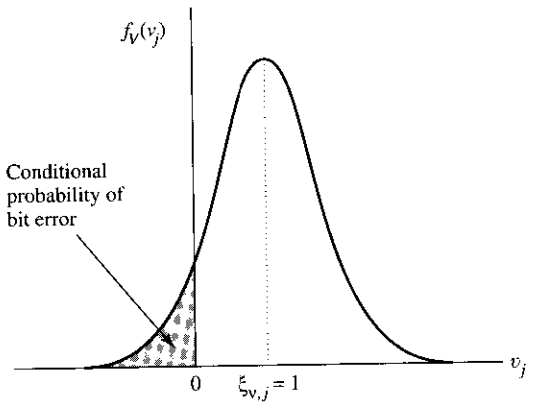


FIGURE 14.15 Conditional probability of bit error, assuming a Gaussian distribution for the induced local field v_j of neuron j; the subscript V in the probability density function $f_v(v_j)$ denotes a random variable with v_j representing a realization of it.

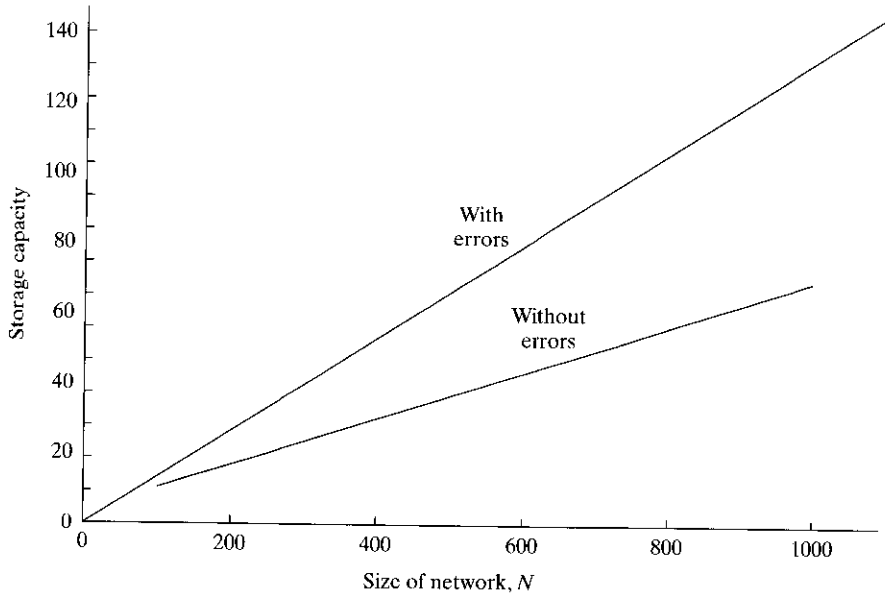


FIGURE 14.16 Plots of storage capacity of the Hopfield network versus network size for two cases: with errors and almost without errors.

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

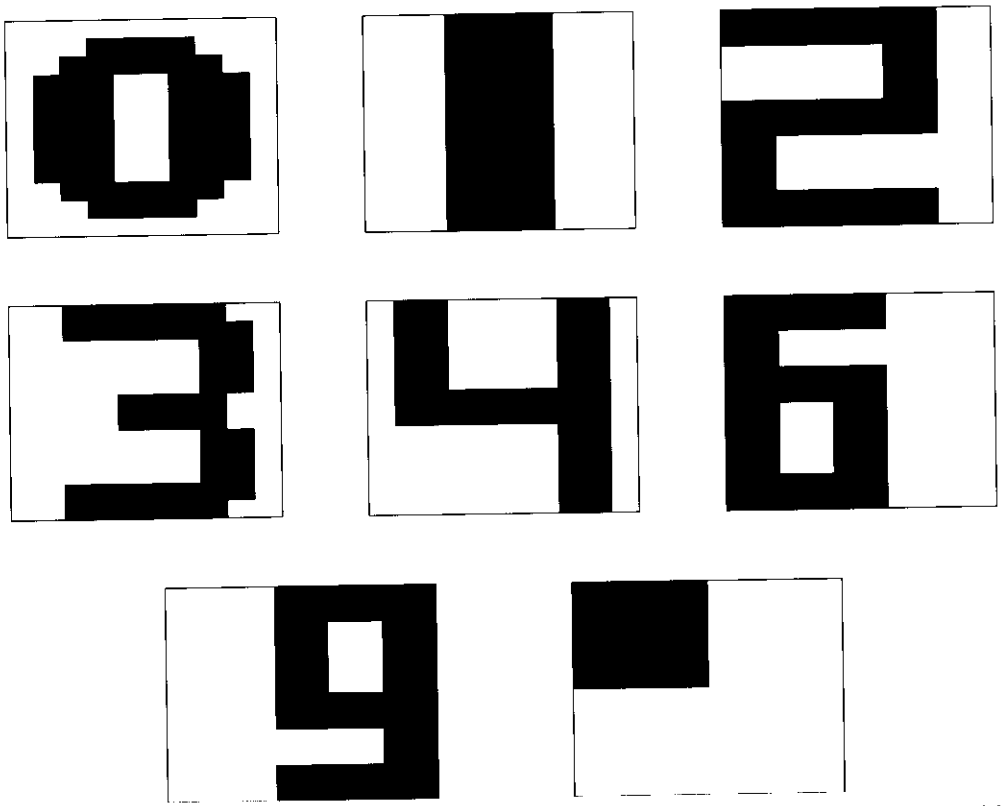


FIGURE 14.17 Set of handcrafted patterns for computer experiment on the Hopfield network.

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

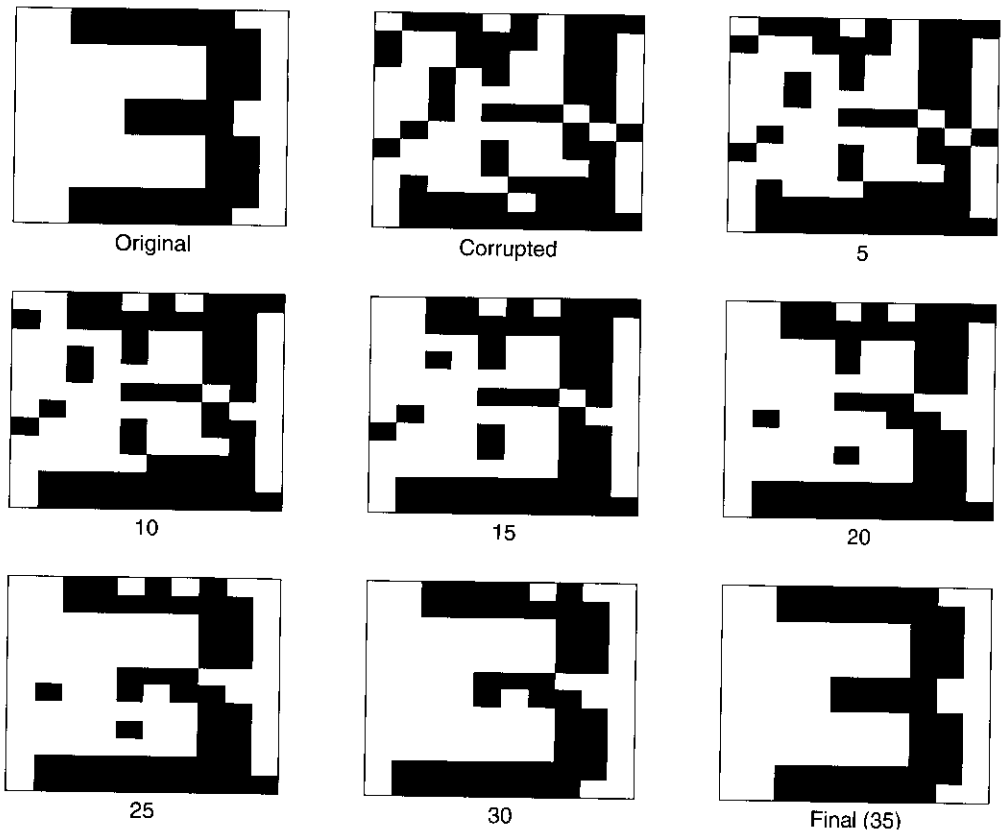


FIGURE 14.18 Correct recollection of corrupted pattern 3.

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

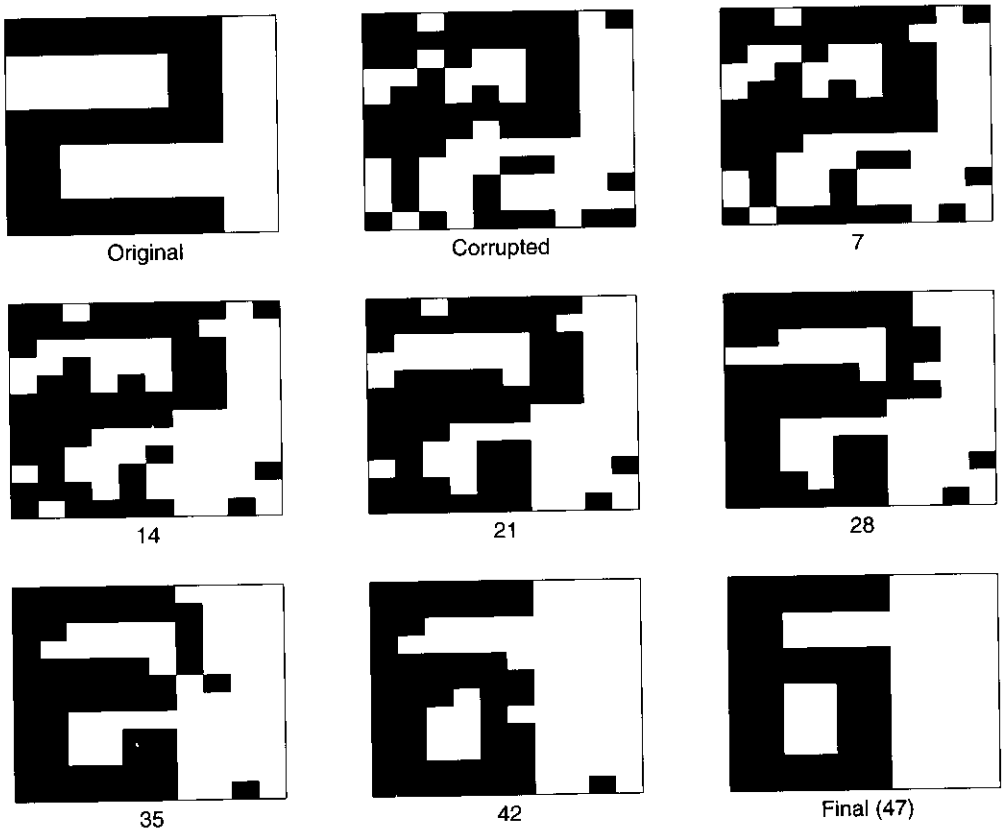


FIGURE 14.19 Incorrect recollection of corrupted pattern 2.

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).

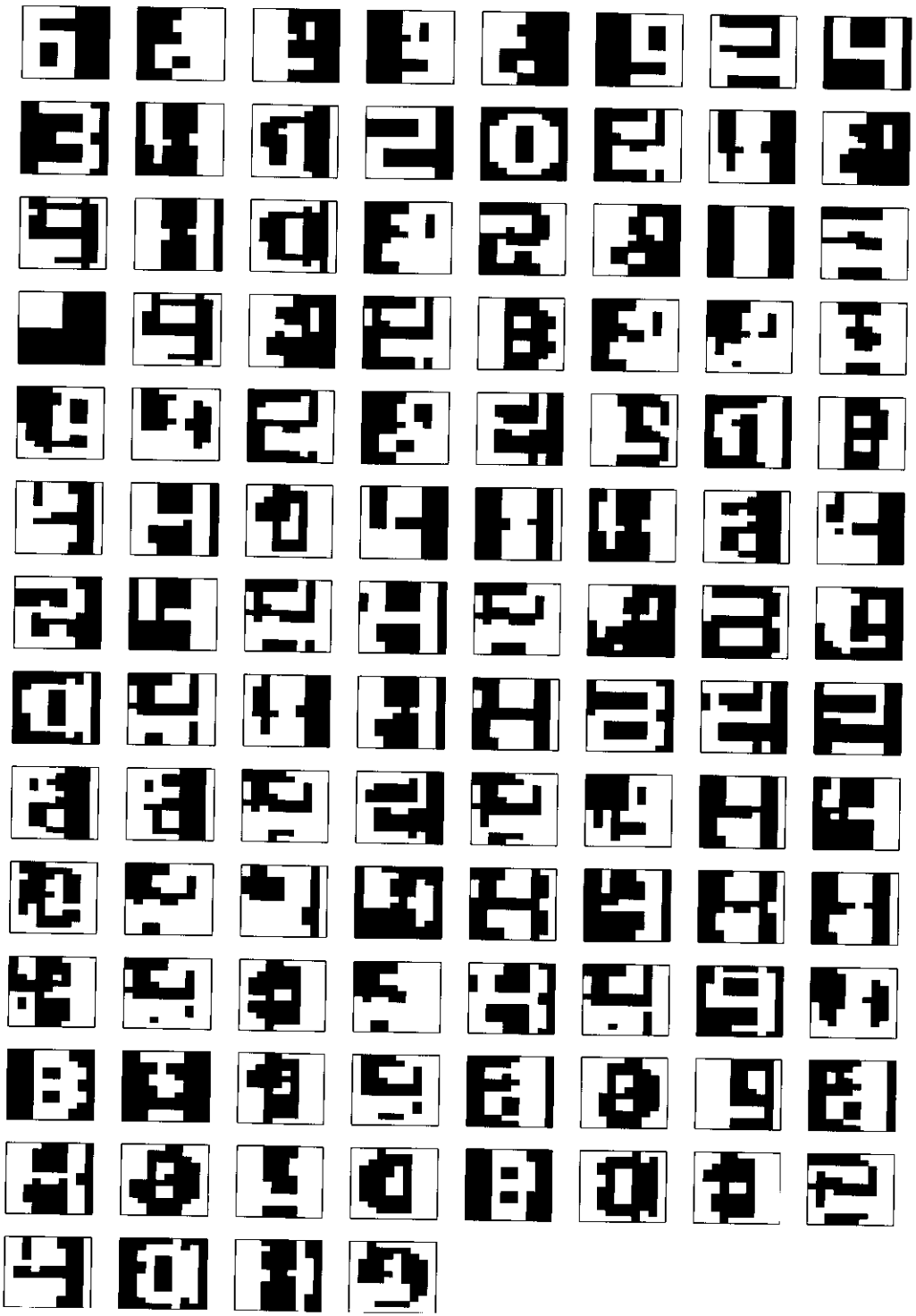


FIGURE 14.20 Compilation of the spurious states produced in the computer experiment on the Hopfield network.

Accordingly, if the fundamental memory ξ_μ corresponds to a particular local minimum of the energy landscape, that same local minimum also corresponds to $-\xi_\mu$. 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 ξ_1, ξ_2, and ξ_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: ξ_1 = negative of digit 1, ξ_2 = digit 4, and ξ_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:"

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

CHAPTER 15

Dynamically Driven Recurrent Networks

15.1 INTRODUCTION

As mentioned in the previous chapter, *recurrent networks* are neural networks with one or more feedback loops. The feedback can be of a *local* or *global* kind. In this chapter we continue the study of recurrent networks with global feedback.

Given a multilayer perceptron as the basic building block, the application of global feedback can take a variety of forms. We may have feedback from the output neurons of the multilayer perceptron to the input layer. Yet another possible form of global feedback is from the hidden neurons of the network to the input layer. When the multilayer perceptron has two or more hidden layers, the possible forms of global feedback expand even further. The point is that recurrent networks have a rich repertoire of architectural layouts.

Basically, there are two functional uses of recurrent networks:

- *Associative memories*
- *Input–output mapping networks*

The use of recurrent networks as associative memories is considered in detail in Chapter 14. In the present chapter, we will study their use as input–output mapping networks. Whatever the use, an issue of particular concern in the study of recurrent networks is that of *stability;* that issue is also considered in Chapter 14.

By definition, the input space of a mapping network is mapped onto an output space. For this kind of an application, a recurrent network responds *temporally* to an externally applied input signal. We may therefore speak of the recurrent networks considered in this chapter as *dynamically driven recurrent networks.* Moreover, the application of feedback enables recurrent networks to acquire *state* representations, which make them suitable devices for such diverse applications as nonlinear prediction and modeling, adaptive equalization of communication channels, speech processing, plant control, and automobile engine diagnostics. As such, recurrent networks offer an alternative to the dynamically driven feedforward networks described in Chapter 13.

Because of the beneficial effects of global feedback, they may actually fare better in these applications. The use of global feedback has the potential of reducing the memory requirement significantly.

Organization of the Chapter

The chapter is organized in four parts: architectures, theory, learning algorithms, and applications. Part 1, consisting of Section 15.2, deals with recurrent network architectures.

Part 2, consisting of Sections 15.3 to 15.5, deals with theoretical aspects of recurrent networks. Section 15.3 describes the state-space model and the related issues of controllability and observability. Section 15.4 derives an equivalent to the state-space model known as the nonlinear autoregressive with exogenous inputs model. Section 15.5 discusses some theoretical issues pertaining to the computational power of recurrent networks.

Part 3, consisting of Sections 15.6 to 15.12, is devoted to learning algorithms and related issues. It starts with an overview of the subject matter in Section 15.6. Then Section 15.7 discusses back-propagation through time that builds on material presented in Chapter 4. Section 15.8 discusses another popular algorithm: real-time recurrent learning. In Section 15.9 we present a brief review of classical Kalman filter theory, followed by a description of the decoupled extended Kalman filtering algorithm in Section 15.10. A computer experiment on this latter algorithm for recurrent learning is presented in Section 15.11. Gradient-based recurrent learning suffers from the vanishing gradients problem, which is discussed in Section 15.12.

The fourth and last part of the chapter, consisting of Sections 15.13 and 15.14, deals with two important applications of recurrent networks. Section 15.13 discusses system identification. Section 15.14 discusses model-reference adaptive control.

The chapter concludes with some final remarks in Section 15.15.

15.2 RECURRENT NETWORK ARCHITECTURES

As mentioned in the introduction, the architectural layout of a recurrent network takes many different forms. In this section we describe four specific network architectures, each of which highlights a specific form of global feedback.[1] They share the following common features:

- They all incorporate a *static* multilayer perceptron or parts thereof.
- They all exploit the nonlinear mapping capability of the multilayer perceptron.

Input–Output Recurrent Model

Figure 15.1 shows the architecture of a generic recurrent network that follows naturally from a multilayer perceptron. The model has a single input that is applied to a tapped-delay-line memory of q units. It has a single output that is fed back to the input via another tapped-delay-line memory also of q units. The contents of these two tapped-delay-line memories are used to feed the input layer of the multilayer perceptron. The present value of the model input is denoted by $u(n)$, and the corresponding

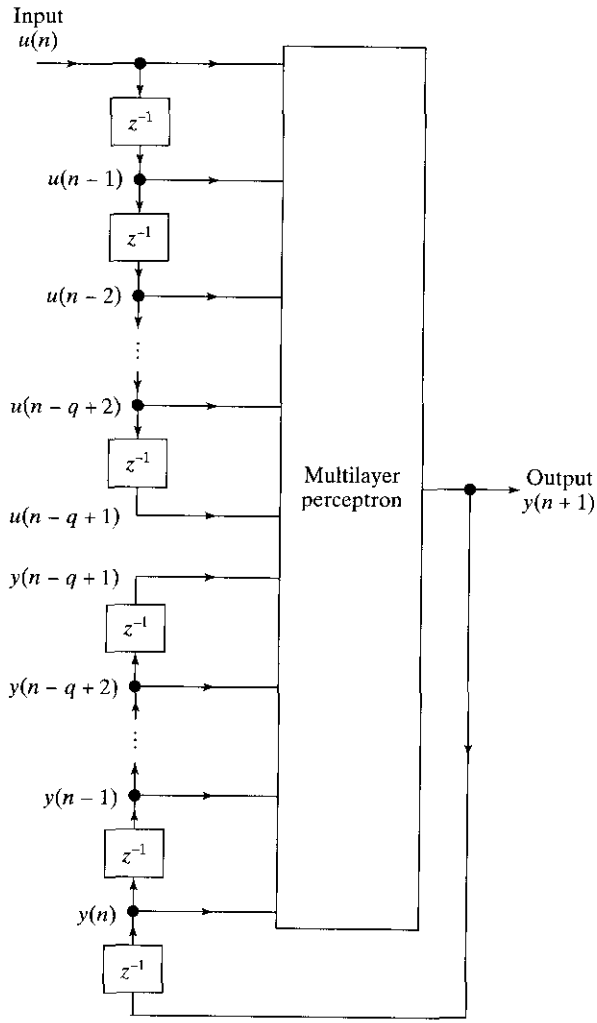


FIGURE 15.1 Nonlinear autoregressive with exogenous inputs (NARX) model.

value of the model output is denoted by $y(n + 1)$; that is, the output is ahead of the input by one time unit. Thus, the signal vector applied to the input layer of the multilayer perceptron consists of a data window made up as follows:

- Present and past values of the input, namely $u(n)$, $u(n - 1)$, ..., $u(n - q + 1)$, which represent *exogenous* inputs originating from outside the network.
- Delayed values of the output, namely, $y(n)$, $y(n - 1)$, ..., $y(n - q + 1)$, on which the model output $y(n + 1)$ is *regressed*.

Thus the recurrent network of Fig. 15.1 is referred to as a *nonlinear autoregressive with exogenous inputs (NARX) model.*[2] The dynamic behavior of the NARX model is described by

$$y(n + 1) = F(y(n), \ldots, y(n - q + 1), u(n), \ldots, u(n - q + 1)) \tag{15.1}$$

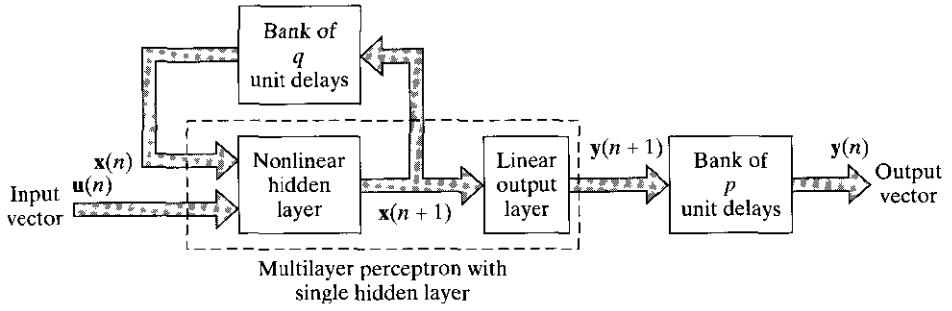


FIGURE 15.2 State-space model.

where F is a nonlinear function of its arguments. Note that in Fig. 15.1 we have assumed that the two delay-line memories in the model are both of size q; they are generally different. The NARX model is explored in greater detail in Section 15.4.

State-Space Model

Figure 15.2 shows the block diagram of another generic recurrent network, called a *state-space model*. The hidden neurons define the *state* of the network. The output of the hidden layer is fed back to the input layer via a bank of unit delays. The input layer consists of a concatenation of feedback nodes and source nodes. The network is connected to the external environment via the source nodes. The number of unit delays used to feed the output of the hidden layer back to the input layer determines the *order* of the model. Let the m-by-1 vector $\mathbf{u}(n)$ denote the input vector, and the q-by-1 vector $\mathbf{x}(n)$ denote the output of the hidden layer at time n. We may then describe the dynamic behavior of the model in Fig. 15.2 by the pair of coupled equations:

$$\mathbf{x}(n+1) = \mathbf{f}(\mathbf{x}(n), \mathbf{u}(n)) \tag{15.2}$$

$$\mathbf{y}(n) = \mathbf{C}\mathbf{x}(n) \tag{15.3}$$

where $\mathbf{f}(\cdot,\cdot)$ is a nonlinear function characterizing the hidden layer, and $\mathbf{C}$ is the matrix of synaptic weights characterizing the output layer. The hidden layer is nonlinear, but the output layer is linear.

The recurrent network of Fig. 15.2 includes several recurrent architectures as special cases. Consider, for example, the *simple recurrent network* (SRN) described in Elman (1990) and depicted in Fig. 15.3. Elman's network has an architecture similar to that of Fig. 15.2 except for the fact that the output layer may be nonlinear and the bank of unit delays at the output is omitted.

Elman's network contains recurrent connections from the hidden neurons to a layer of *context units* consisting of unit delays. These context units store the outputs of the hidden neurons for one time step, and then feed them back to the input layer. The hidden neurons thus have some record of their prior activations, which enables the network to perform learning tasks that extend over time. The hidden neurons also feed the output neurons that report the response of the network to the externally applied stimulus. Due to the nature of feedback around the hidden neurons, these neurons may

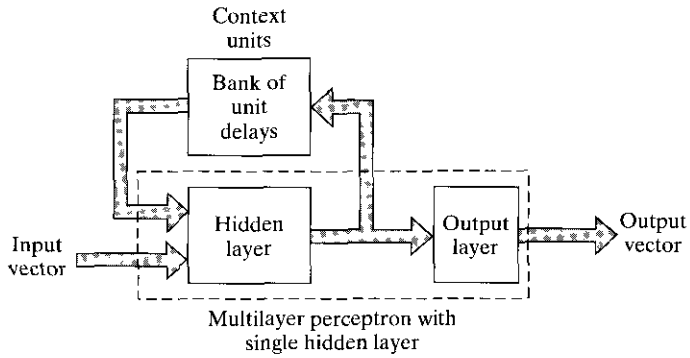


FIGURE 15.3 Simple Recurrent network (SRN).

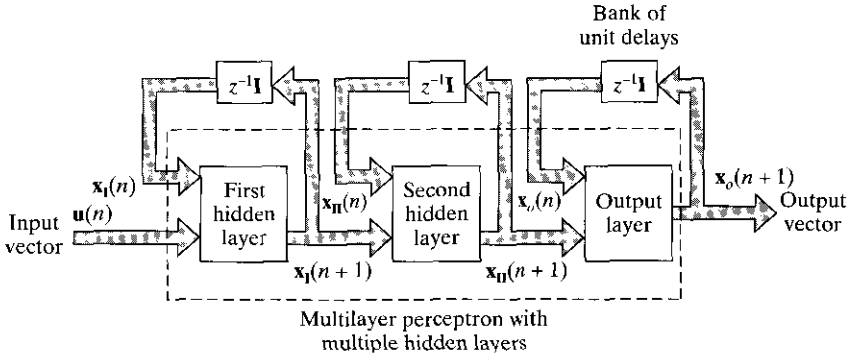


FIGURE 15.4 Recurrent multilayer perceptron.

continue to recycle information through the network over multiple time steps, and thereby discover abstract representations of time. The simple recurrent network is therefore not merely a tape recording of past data.

Elman (1990) discusses the use of the simple recurrent network shown in Fig. 15.3 to discover word boundaries in a continuous stream of phonemes without any built-in representational constraints. The input to the network represents the current phoneme. The output represents the network's best guess as to what the next phoneme is in the sequence. The role of the context units is to provide the network with *dynamic memory* so as to encode the information contained in the sequence of phonemes, which is relevant to the prediction.

Recurrent Multilayer Perceptron

The third recurrent architecture considered here is known as a *recurrent multilayer perceptron* (RMLP) (Puskorius et al., 1996). It has one or more hidden layers, basically for the same reasons that static multilayer perceptrons are often more effective and parsimonious than those using a single hidden layer. Each computation layer of an RMLP has feedback around it, as illustrated in Fig. 15.4 for the case of an RMLP with two hidden layers.[3]

Let the vector $\mathbf{x}_{\mathrm{I}}(n)$ denote the output of the first hidden layer, $\mathbf{x}_{\mathrm{II}}(n)$ denote the output of the second hidden layer, and so on. Let the vector $\mathbf{x}_o(n)$ denote the output of the output layer. Then the dynamic behavior of the RMLP, in general, in response to an input vector $\mathbf{u}(n)$ is described by the following system of coupled equations:

$$\begin{aligned}
\mathbf{x}_{\mathrm{I}}(n+1) &= \boldsymbol{\varphi}_{\mathrm{I}}(\mathbf{x}_{\mathrm{I}}(n), \mathbf{u}(n)) \\
\mathbf{x}_{\mathrm{II}}(n+1) &= \boldsymbol{\varphi}_{\mathrm{II}}(\mathbf{x}_{\mathrm{II}}(n), \mathbf{x}_{\mathrm{I}}(n+1)) \\
&\vdots \\
\mathbf{x}_o(n+1) &= \boldsymbol{\varphi}_o(\mathbf{x}_o(n), \mathbf{x}_K(n+1))
\end{aligned} \tag{15.4}$$

where $\boldsymbol{\varphi}_{\mathrm{I}}(\cdot,\cdot), \boldsymbol{\varphi}_{\mathrm{II}}(\cdot,\cdot), \ldots, \boldsymbol{\varphi}_o(\cdot,\cdot)$ denote the activation functions characterizing the first hidden layer, second hidden layer, ..., and output layer of the RMLP, respectively; and K denotes the number of hidden layers in the network.

The RMLP described herein subsumes the Elman network of Fig. 15.3 and the state-space model of Fig. 15.2 since the output layer of the RMLP or any of its hidden layers is not constrained to have a particular form of activation function.

Second-Order Network

In describing the state-space model of Fig. 15.2 we used the term "order" to refer to the number of hidden neurons whose outputs are fed back to the input layer via a bank of unit delays.

In yet another context, the term "order" is sometimes used to refer to the way in which the induced local field of a neuron is defined. Consider, for example, a multilayer perceptron where the induced local field v_k of neuron k is defined by

$$v_k = \sum_j w_{a,kj} x_j + \sum_i w_{b,ki} u_i \tag{15.5}$$

where x_j is the feedback signal derived from hidden neuron j and u_i is the source signal applied to node i in the input layer; the w's represent the pertinent synaptic weights in the network. We refer to a neuron described in Eq. (15.5) as a *first-order neuron*. When, however, the induced local field v_k is combined using multiplications, as shown by

$$v_k = \sum_i \sum_j w_{kij} x_i u_j \tag{15.6}$$

we refer to the neuron as a *second-order neuron*. The second-order neuron k uses a single weight, w_{kji}, that connects it to the input nodes i and j.

Second-order neurons constitute the basis of *second-order recurrent networks* (Giles et al., 1990), an example of which is shown in Fig. 15.5. The network accepts a time-ordered sequence of inputs and evolves with dynamics defined by the following pair of equations:

$$v_k(n) = b_k + \sum_i \sum_j w_{kij} x_i(n) u_j(n) \tag{15.7}$$

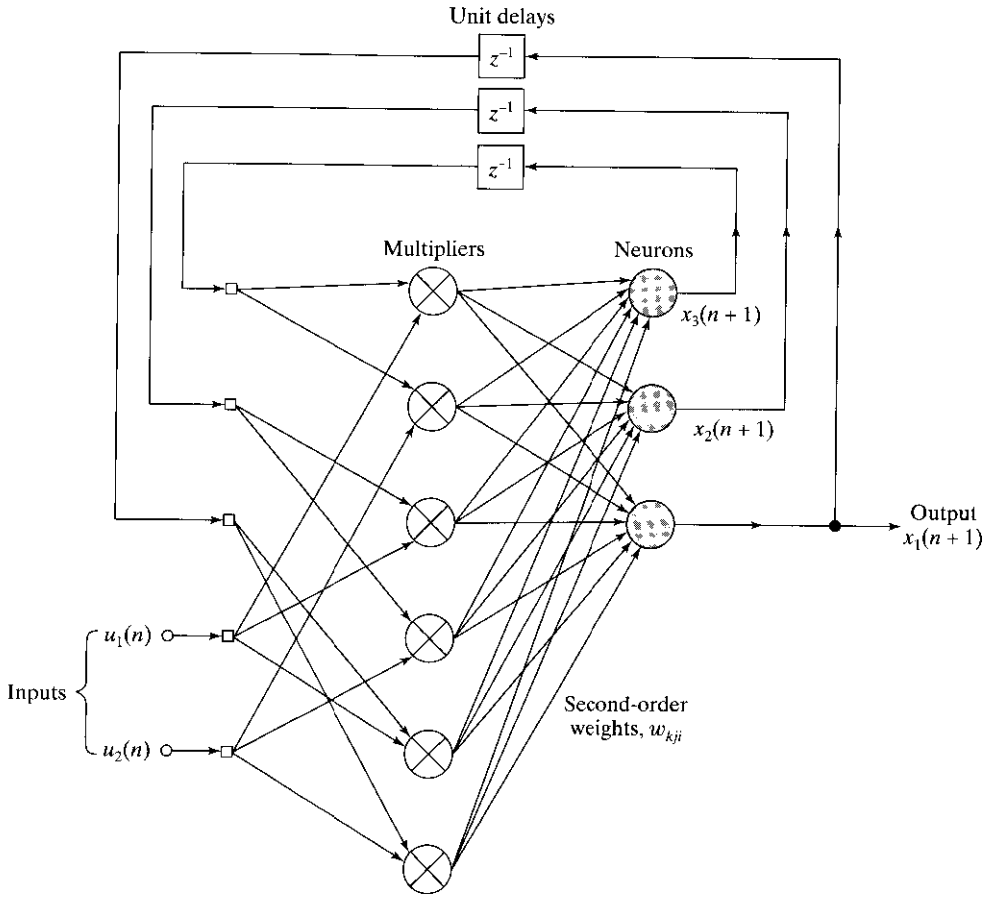


FIGURE 15.5 Second-order recurrent network; bias connections to the neurons are omitted to simplify the presentation. The network has 2 inputs and 3 state neurons, hence the need for 3 x 2 = 6 multipliers.

and

$$\begin{aligned} x_k(n+1) &= \varphi(v_k(n)) \\ &= \frac{1}{1+\exp(-v_k(n))} \end{aligned} \tag{15.8}$$

where $v_k(n)$ is the induced local field of hidden neuron k, b_k is the associated bias, $x_k(n)$ is the state (output) of neuron k, $u_j(n)$ is the input applied to source node j, and w_{kij} is a weight of second-order neuron k.

A unique feature of the second-order recurrent network in Fig. 15.5 is that the product $x_j(n)u_j(n)$ represents the pair {state, input} and that a positive weight w_{kij} represents the presence of the state transition, {state, input} $\rightarrow$ {next state}, while a negative weight represents the absence of the transition. The state transition is described by

$$\delta(x_i, u_j) = x_k \tag{15.9}$$

In light of this relationship, second-order networks are readily used for representing and learning *deterministic finite-state automata*[4] (DFA); a DFA is an information processing device with a finite number of states. More information on the relationship between neural networks and automata is found in Section 15.5.

The recurrent network architectures discussed in this section emphasize the use of global feedback. As mentioned in the introduction, it is also possible for a recurrent network architecture to have only local feedback. A summary of the properties of this latter class of recurrent networks is presented in Tsoi and Back (1994); see also Problem 15.7.

15.3 STATE-SPACE MODEL

The notion of *state* plays a vital role in the mathematical formulation of a dynamical system. The state of a dynamical system is formally defined as a *set of quantities that summarizes all the information about the past behavior of the system that is needed to uniquely describe its future behavior, except for the purely external effects arising from the applied input (excitation).* Let the q-by-1 vector $\mathbf{x}(n)$ denote the state of a nonlinear discrete-time system. Let the m-by-1 vector $\mathbf{u}(n)$ denote the input applied to the system, and the p-by-1 vector $\mathbf{y}(n)$ denote the corresponding output of the system. In mathematical terms, the dynamic behavior of the system, assumed to be *noise free,* is described by the following pair of nonlinear equations (Sontag, 1996):

$$\mathbf{x}(n+1) = \boldsymbol{\varphi}(\mathbf{W}_a\mathbf{x}(n) + \mathbf{W}_b\mathbf{u}(n)) \tag{15.10}$$

$$\mathbf{y}(n) = \mathbf{C}\,\mathbf{x}(n) \tag{15.11}$$

where $\mathbf{W}_a$ is a q-by-q matrix, $\mathbf{W}_b$ is a q-by-$(m+1)$ matrix, $\mathbf{C}$ is a p-by-q matrix; and $\boldsymbol{\varphi} : \mathbb{R}^q \to \mathbb{R}^q$ is a diagonal map described by

$$\boldsymbol{\varphi}: \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_q \end{bmatrix} \to \begin{bmatrix} \varphi(x_1) \\ \varphi(x_2) \\ \vdots \\ \varphi(x_q) \end{bmatrix} \tag{15.12}$$

for some memoryless, component-wise nonlinearity $\varphi: \mathbb{R} \to \mathbb{R}$. The spaces $\mathbb{R}^m$, $\mathbb{R}^q$, and $\mathbb{R}^p$ are called the *input space, state space,* and *output space,* respectively. The dimensionality of the state space, namely q, is the *order* of the system. Thus the state-space model of Fig. 15.2 is an *m-input, p-output recurrent model of order q.* Equation (15.10) is the *process equation* of the model and Eq. (15.11) is the *measurement equation.* The process equation (15.10) is a special form of Eq. (15.2).

The recurrent network of Fig. 15.2, based on the use of a static multilayer perceptron and two delay-line memories, provides a method for implementing the nonlinear feedback system described by Eqs. (15.10) to (15.12). Note that in Fig. 15.2 *only those neurons in the multilayer perceptron that feed back their outputs to the input layer via delays are responsible for defining the state of the recurrent network.* This statement therefore excludes the neurons in the output layer from the definition of the state.

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

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

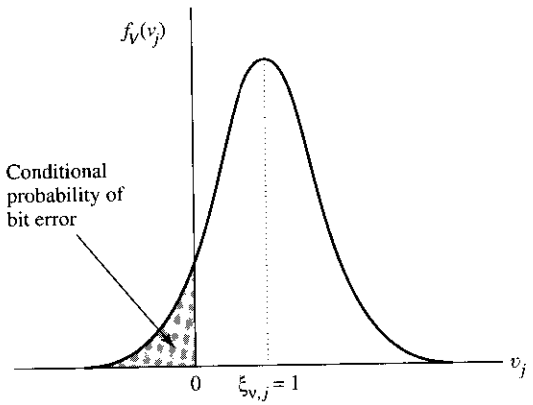


FIGURE 14.15 Conditional probability of bit error, assuming a Gaussian distribution for the induced local field v_j of neuron j; the subscript V in the probability density function $f_v(v_j)$ denotes a random variable with v_j representing a realization of it.

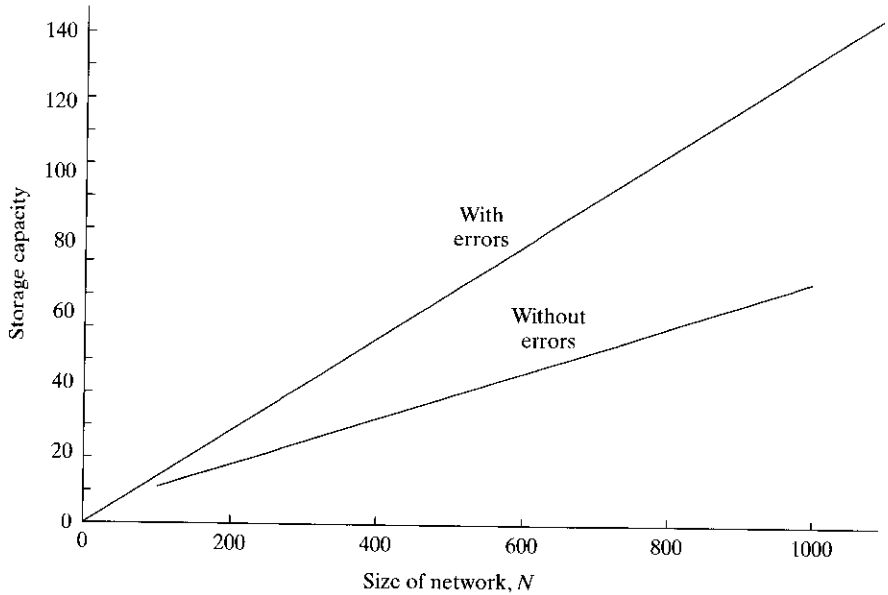


FIGURE 14.16 Plots of storage capacity of the Hopfield network versus network size for two cases: with errors and almost without errors.

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

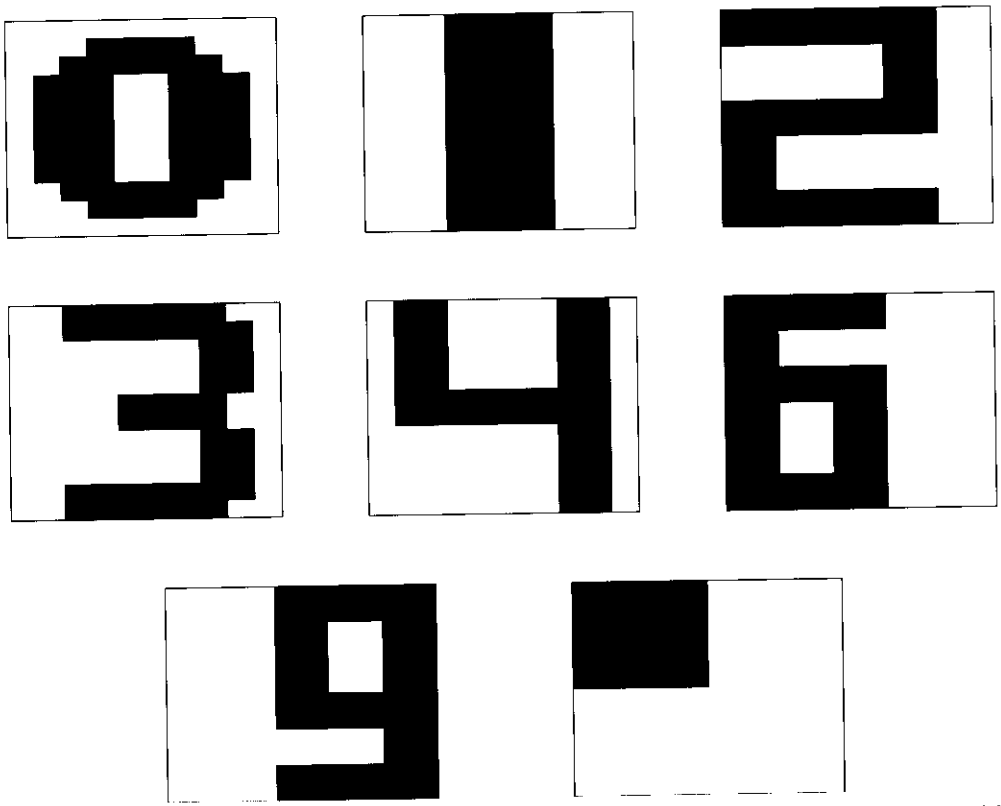


FIGURE 14.17 Set of handcrafted patterns for computer experiment on the Hopfield network.

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

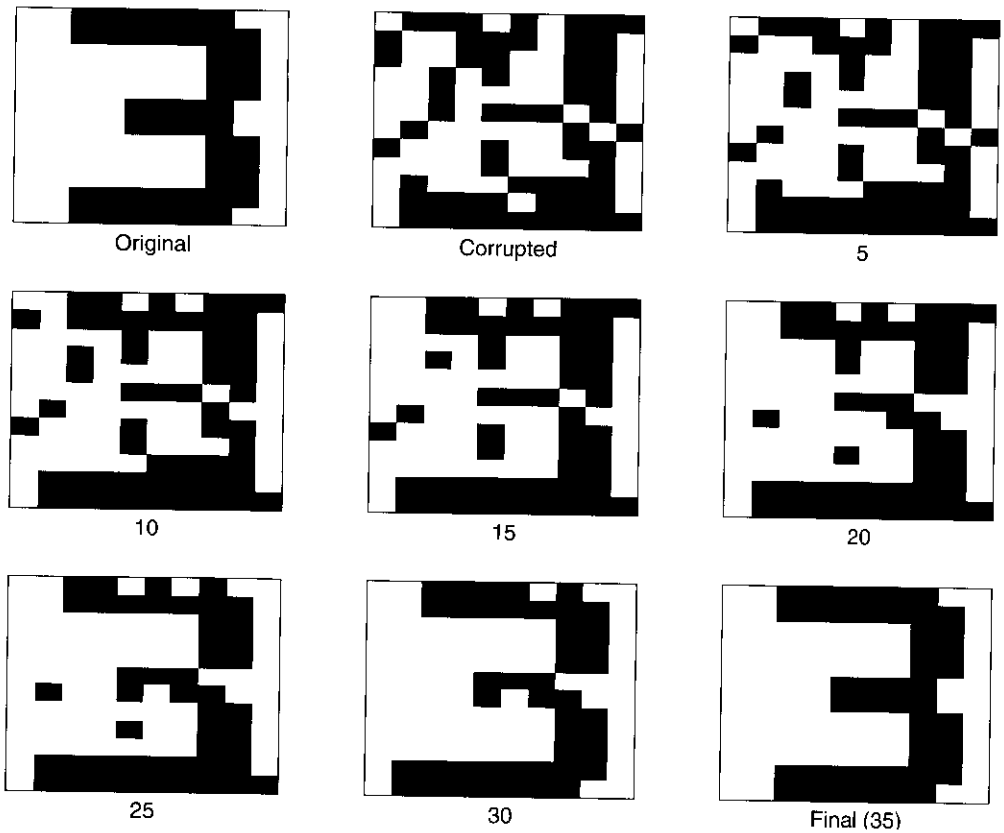


FIGURE 14.18 Correct recollection of corrupted pattern 3.

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

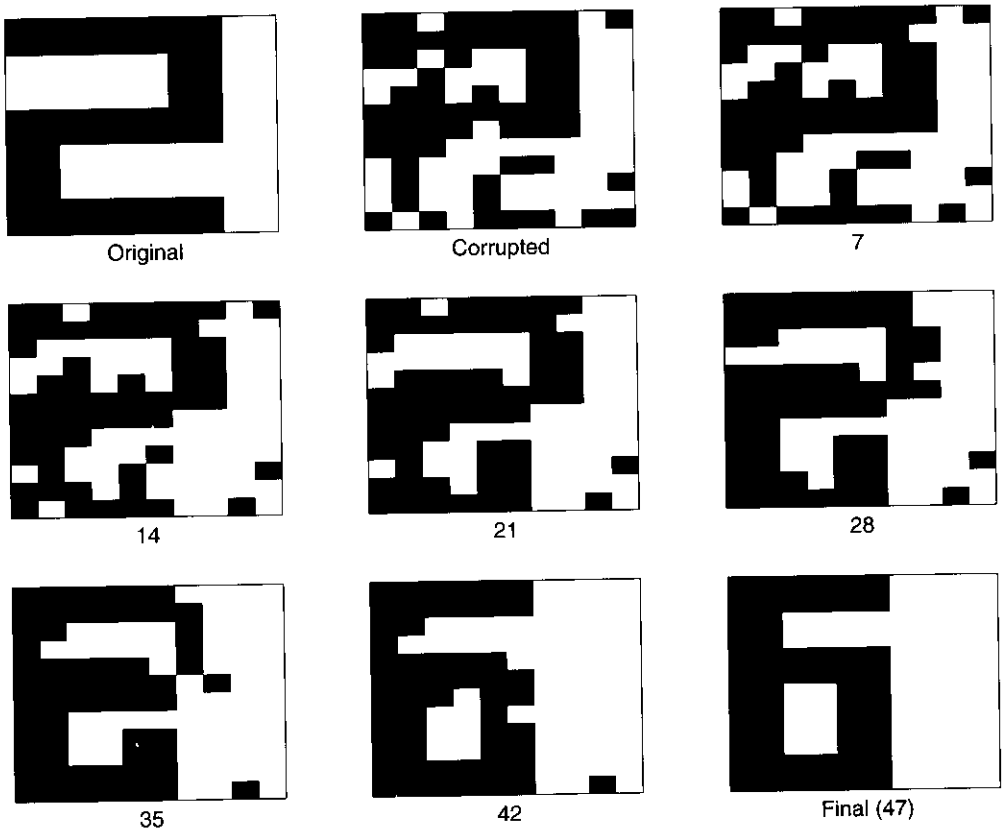


FIGURE 14.19 Incorrect recollection of corrupted pattern 2.

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).

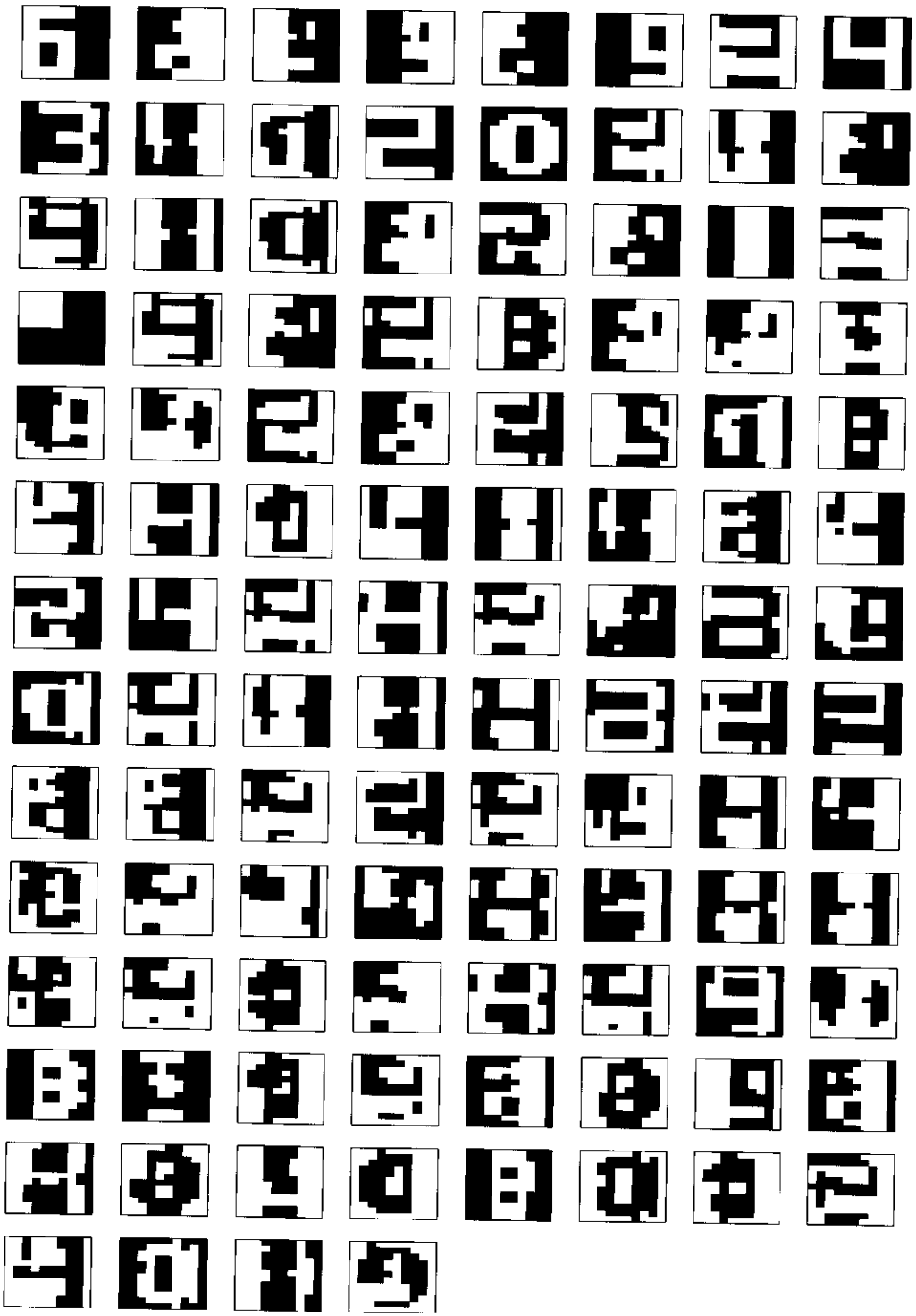


FIGURE 14.20 Compilation of the spurious states produced in the computer experiment on the Hopfield network.

Accordingly, if the fundamental memory ξ_μ corresponds to a particular local minimum of the energy landscape, that same local minimum also corresponds to $-\xi_\mu$. 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 ξ_1, ξ_2, and ξ_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: ξ_1 = negative of digit 1, ξ_2 = digit 4, and ξ_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:"

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

for estimating σ^2. A standard estimate for this term [93] is:

$$\hat{\sigma}^2 = \left(\frac{P}{P - N_w}\right) MSE \tag{36}$$

Recently, Moody has proposed a similar formula that is applicable to systems which are trained using a complexity regularization term [93, 94]. This formula is called the *generalized prediction error* (GPE), and is of the form:

$$GPE = MSE + \frac{2N_{eff}}{P}\sigma^2 \tag{37}$$

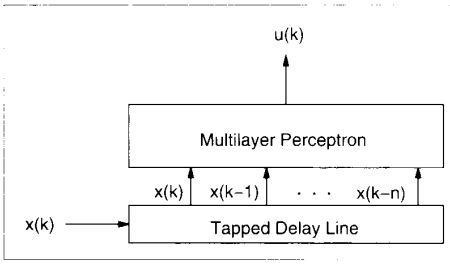
where N_{eff} is the *effective number of parameters* in the network. Because of complexity regularization, the effective number of parameters is typically much less than the actual number of parameters. Methods for estimating N_{eff} are discussed in [93, 94]. The corresponding estimate of the noise variance is the same as in Eq. 36, with N_w replaced by N_{eff}.

Dynamic Networks

The second class of networks we will discuss are dynamic networks. The node equations in these networks are described by differential or difference equations. These networks are important because many of the systems that we wish to model in the real world are nonlinear dynamical systems. This is true, for example, in the controls area in which we wish to model the forward or inverse dynamics of systems such as airplanes, rockets, spacecraft, and robots. Another class of nonlinear systems we wish to model are finite state machines which are at the heart of all modern day computers. Although these two examples may seem very different in nature, the networks discussed in this section can be used to model both. In this sense, they are very general models, with the potential for use in a wide variety of applications.

Time Delay Neural Network (TDNN)

Before we discuss networks that are truly dynamic, consider how an MLP is often used to process time series data. It is possible to use a static network to process time series data by simply converting the temporal sequence into a static pattern by unfolding the sequence over time. That is, time is treated as another dimension in the problem. From a practical point of view we can only afford to unfold the sequence over a finite period of time.



12. Time delay neural network.

This can be accomplished by feeding the input sequence into a tapped delay line of finite extent, then feeding the taps from the delay line into a static neural network architecture like a Multilayer Perceptron (Fig. 12). An architectures like this is often referred to as a Time Delay Neural Network (TDNN) [51]. It is capable of modeling systems where the output has a finite temporal dependence on the input, that is:

$$u(k) = F[x(k), x(k-1), ..., x(k-n)] \tag{38}$$

When the function $F(\cdot)$ is a weighted linear sum, this architecture is equivalent to a linear *finite impulse response* (FIR) filter. Because there is no feedback in this network, it can be trained using the standard Backpropagation algorithm.

The TDNN has been used quite successfully in many applications. The celebrated NETtalk project [124] used the TDNN for text-to-speech conversion. In this system, the input consisted of a local encoding of the alphabet and a small number of punctuation symbols. The output of the network was trained to give the appropriate articulary parameters to a commercial speech synthesizer. These signals represented the phoneme to be uttered at the point of text corresponding to the center character in the tapped-delay line. A version of the TDNN with weight sharing and local connections has been used for speech recognition with excellent results [73]. The TDNN has also been applied to nonlinear time series prediction problems [74]. The same approach using a Radial Basis Function Network in place of Multilayer Perceptrons was investigated in [92].

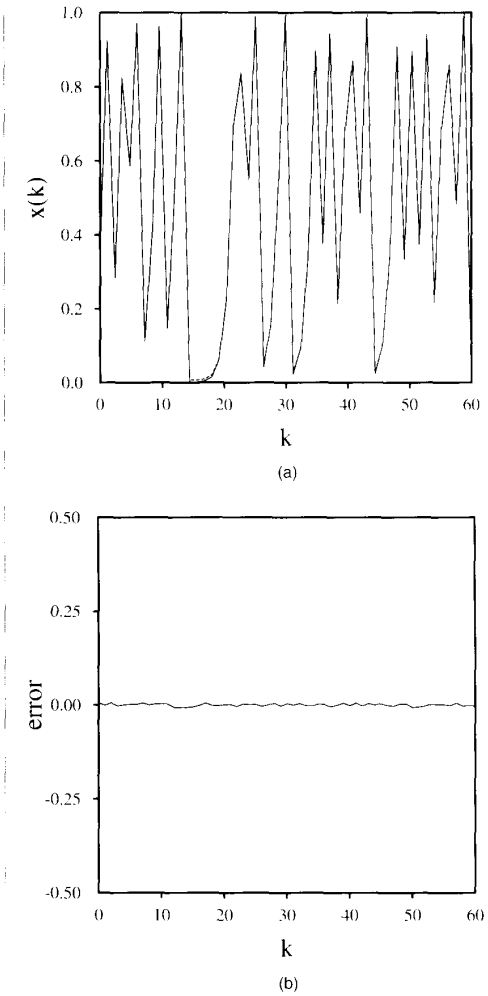
As an example, we trained the TDNN to perform 1-step prediction of the chaotic sequence:

$$x(k) = 4.0\, x(k-1)\,[1.0 - x(k-1)] \tag{39}$$

A two layer network was used with 2 hidden layer nodes. The input to the network was of order 1, i.e., only one delay was used in the tapped delay line. Figure 13 shows the actual and predicted sequences after training (the two are virtually indistinguishable).

Networks with Feedback Dynamics

Dynamical systems with feedback can offer great advantages over purely feedforward systems. For some problems, a small feedback system is equivalent to a large and possibly infinite feedforward system. For example, it is well known that an infinite number of feedforward logic gates are required to emulate an arbitrary finite state machine, or that an infinite order FIR filter is required to emulate a single pole infinite impulse response (IIR) filter [104, 130]. Systems with feedback are particularly appropriate for identification (modeling), control, and filtering applications. Most of the conventional work in these areas has been dominated by linear systems theory. But, as mentioned above, there are many problems which require nonlinear dynamics. Although the neural networks discussed here are by no means the only



13. Prediction of the logistic equation with the TDNN. Chaotic sequence and its predicted value (a); error signal (b).

approach to dynamical systems, they are interesting because of their neurological motivation and because of their applicability to a diverse set of nonlinear problems. This class of networks are commonly referred to as recurrent neural networks because they incorporate feedback and thus are inherently recursive.

One difficulty with recurrent networks is developing meaningful learning algorithms. For the most part, these learning algorithms are gradient search techniques similar to Backpropagation for Multilayer Perceptrons. Since the output of the nodes is a recursive function of the output of nodes on the previous time step, the calculation of the gradient must also be a recursive computation. This makes these learning algorithms considerably more complex.

Networks with Output Feedback

A simple way to incorporate feedback into a neural network architecture is to feed back the output of the network through a second tapped-delay line (Fig. 14). This particular architecture was introduced by Narendra [98] and has been used primarily for nonlinear identification and control problems [95]. This architecture is very general. In fact, if we apply the mapping theorems previously discussed for MLPs, then this architecture is capable, in theory, of modeling any system which can be expressed as:

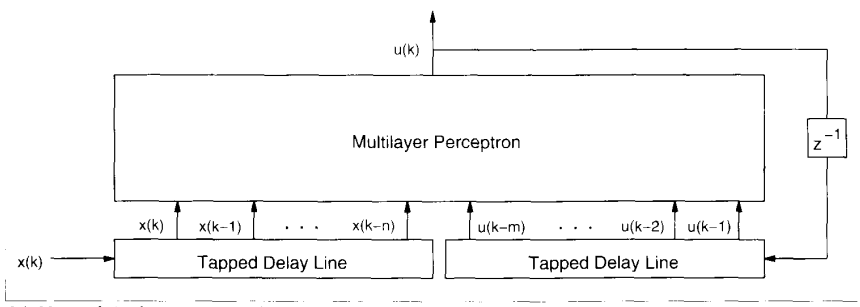
$$u(k)=F[x(k),x(k-1),\ldots x(k-n),u(k-1),u(k-2),\ldots,u(k-m)] \tag{40}$$

When the function $F(\cdot)$ is replaced by a weighted linear sum, this architecture is equivalent to an IIR filter. A constrained version of this architecture has also been proposed in which the outputs of the two tapped-delay lines are processed by two separate networks whose individual outputs are then summed to form the overall network output [57, 98]. The development of gradient descent learning algorithms for this architecture is beyond the scope of this article; however, the reader is referred to [57, 99] for more information.

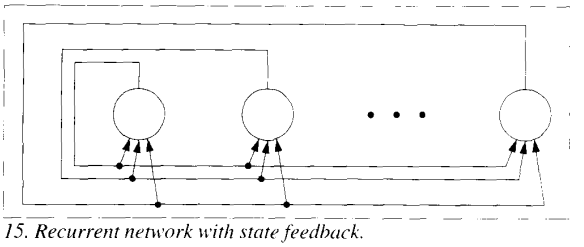
Networks with State Feedback

The next class of dynamic network models that we discuss incorporates a different type of feedback called state feedback. These networks are typically single-layer networks with feedback connections between nodes. In the most general case the nodes are completely interconnected, i.e., every node is connected to every other node, and also to itself (Fig. 15). Every node in the network contributes one component to the state vector. Any or all of the node outputs can be viewed as outputs of the network.

Additionally, any or all of these nodes may receive external inputs. This class of networks is perhaps the most general since many of the networks in previous sections can be obtained as simplifications of these.



14. Narenda's dynamic neural network.



15. Recurrent network with state feedback.

Continuous-Time Hopfield Net

The Hopfield network is probably the best known dynamic network model [53, 54]. It is a single-layer network with complete interconnections. The node equations for the continuous-time Hopfield network are given by:

$$\tau_i \dot{y}_i(t) = -y_i(t) + \sum_{j=1}^{N} w_{i,j} u_j(t) + v_i \qquad (41)$$
$$u_i(t) = f(y_i(t))$$

where $y_i(t)$ is the *internal state* of the ith neuron, $u_i(t)$ is the *output activation* (or *output state*) of the ith neuron, $w_{i,j}$ is the weight connecting the jth neuron to the ith neuron, and v_i is the input to the ith neuron.

The Hopfield network can be viewed as a nonlinear dynamical system with input vector $\boldsymbol{v}$, state vector $\boldsymbol{y}(t)$, and output vector $\boldsymbol{u}(t)$ as shown in Fig. 16. Because of the sigmoid nonlinearity, the output vector lies in the interior of an N-dimensional unit hypercube; that is, $\boldsymbol{u}(t) \in (0,1)^N$.

The Hopfield network is a nonlinear dynamical system which is capable of exhibiting a wide range of complex behavior. Depending on how the network parameters are chosen, it may function as a stable system, an oscillator, or even a chaotic system [2, 22, 32, 122]. Most of Hopfield's original applications required that the network perform as a stable system with multiple asymptotically stable equilibrium points. Conditions which guarantee this type of behavior are described below.

The asymptotic stability of a Hopfield network can be shown using Lyapunov's second method [137]. This method basically works by showing that the system is dissipating energy with time. This proof is accomplished by forming a Lyapunov function (or energy function) for the network, and showing that its time derivative is nonincreasing. The Lyapunov function for this network is defined as follows [23, 54]

$$E(t) = -\frac{1}{2}\boldsymbol{u}(t)^T \boldsymbol{W}\boldsymbol{u}(t) - \boldsymbol{u}(t)^T \boldsymbol{v} + \sum_{i=1}^{n} \int_0^{u_i(t)} f^{-1}(\alpha)\, d\alpha \qquad (42)$$

where $f(\cdot)$ is defined in Eq. 2. Note that Eq. 42 is not a typical Lyapunov function because it can take on negative values. However, it is easy to show that this function is bounded below, and can thus be made positive with an additive constant. This fact is sufficient for the proof of stability. Taking the time derivative of Eq. 42 yields:

$$\dot{E}(t) = \frac{dE(t)}{dt} = (\nabla_{\boldsymbol{u}} E(t))^T \dot{\boldsymbol{u}}(t) \qquad (43)$$

where $\nabla_{\boldsymbol{u}}$ is the gradient operator with respect to $\mathbf{u}$.

With symmetric $\mathbf{W}$, we have:

$$\nabla_{\boldsymbol{u}} E(t) = -\boldsymbol{W}\boldsymbol{u}(t) - \boldsymbol{v} + \boldsymbol{y}(t) \qquad (44)$$

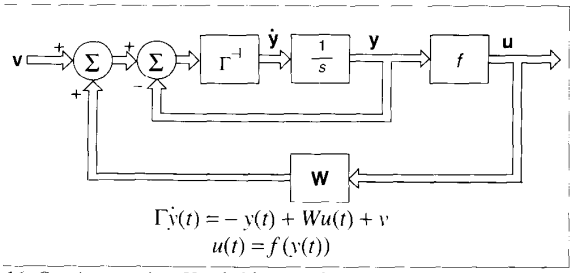
which from the equation in Fig. 16 is simply:

$$\nabla_{\boldsymbol{u}} E(t) = -\Gamma \dot{\boldsymbol{y}}(t) \qquad (45)$$

With this, $\dot{E}(t)$ in Eq. 43 becomes:

$$\dot{E}(t) = -\dot{\boldsymbol{y}}(t)^T \Gamma \dot{\boldsymbol{u}}(t) \qquad (46)$$
$$= -\sum_{i=1}^{n} \tau_i \dot{y}_i(t)\, \dot{u}_i(t) = -\sum_{i=1}^{n} \tau_i \dot{y}_i(t)^2 \frac{\partial u_i}{\partial y_i}$$

Since τ_i and $\dot{y}_i(t)^2$ are always positive, we need only show that $\partial u_i / \partial y_i \geq 0$ to show that $\dot{E}(t) \leq 0$. This is trivially true, however, since $f(\cdot)$ in Eq. 2 is monotonically increasing. We also note from Eq. 46 that $\dot{E}(t)=0$ only when $\dot{\boldsymbol{y}}(t) = \mathbf{0}$. This completes the proof of asymptotic stability. Thus, under the condition that $\mathbf{W}$ is symmetric, the network will eventually reach a *fixed equilibrium point*. Furthermore, the locations of these equilibrium points, which can be found by setting $\dot{\boldsymbol{y}}(t)= 0$ in Fig. 16, are also extrema of E(t). This is easily verified from the above equations by noting that the extrema of $E(t)$ are defined by Eq. 45, with $\dot{\boldsymbol{y}}(t)= \mathbf{0}$.



16. Continuous-time Hopfield network.

The above result tells us that given any set of initial conditions $\boldsymbol{u}(0)$, the Hopfield network (with symmetric $\boldsymbol{W}$) will converge to a fixed equilibrium point, that is, to a point where $\dot{\boldsymbol{u}}(t)=0$. This equilibrium point, $\mathbf{u}_f$, is a fixed point in $(0,1)^n$. Because the network is deterministic, the location of this point is uniquely determined by the initial conditions. That is, the nonlinear nature of the Hopfield network gives rise to multiple equilibrium points, and the one chosen on any particular run of the network is determined uniquely by the initial conditions. All initial conditions that fall within the region of attraction of an equilibrium point will asympotically converge to that point. The exact number of equilibrium points, and their locations, are determined by the network parameters $\mathbf{W}$, $\mathbf{v}$, and β. At low gain (β small) the number of

equilibrium points is small (possibly as small as 1), and their locations lie towards the interior of the hypercube. As the gain is increased, however, the number of equilibrium points grows large and their locations move towards the corners of the hypercube [54].

In the high-gain limit (β approaches infinity) the equilibrium points actually reach the corners of the hypercube and are maximum in number. This number is at most exponential in N [1, 87]. In addition, as β approaches infinity, one can show that the third term in Eq. 42 approaches zero, so that the energy function for the network simplifies to [54]:

$$E(t) = -\frac{1}{2}\boldsymbol{u}(t)^T \boldsymbol{W}\boldsymbol{u}(t) - \boldsymbol{u}(t)^T \boldsymbol{v} \tag{47}$$

The high-gain characteristics of the Hopfield network are of interest because many of the problems that we solve with this network require binary solutions; that is, we wish the equilibrium points, $\mathbf{u}_f$, to be binary vectors.

Once the gain is fixed (possibly at infinity so that hard-limiting nonlinearities are used), the locations of the equilibrium points are determined by $\mathbf{W}$ and $\mathbf{v}$. When used to solve problems like the associative memory problem discussed below, the challenge is to design $\mathbf{W}$ and $\mathbf{v}$ so that the equilibrium points of the network correspond to solutions of the problem.

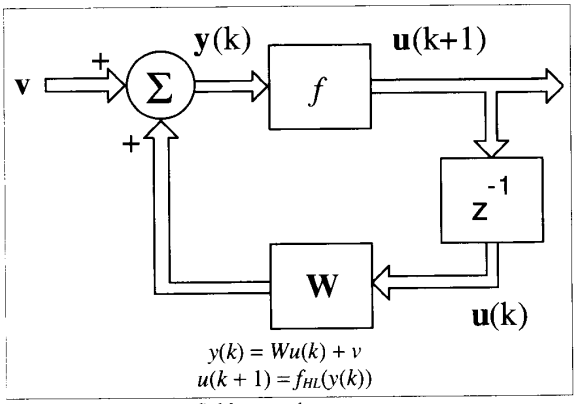
Discrete-Time Hopfield Network

A popular discrete-time version of the Hopfield network is described by the following node equations:

$$\begin{aligned} y_i(k) &= \sum_{j=1}^{N} w_{ij} u_j(k) + v_i \\ u_i(k+1) &= f_{HL}(y_i(k)) \end{aligned} \tag{48}$$

where k is the time increment. These are a discrete-time approximation to Eq. 41, with the sigmoids replaced by hard-limiters. The behavior of this system is similar to Eq. 41: given any set of initial conditions $\mathbf{u}(0)$, and appropriate restrictions on the weights (given below), this network will converge to a fixed equilibrium point. Because a hard-limiting nonlinearity is used, these equilibrium points are binary vectors that are minima of Eq. 47. The system diagram for this network is shown in Fig. 17.

The stability and convergence properties of the discrete-time model are discussed in [18]. Sufficient conditions for stability are that $\mathbf{W}$ be symmetric and positive definite. Actually these conditions depend on the type of update that is used in the node equations. If *synchronous* updates are used, which implies that all nodes are updated simultaneously as suggested Fig. 17, then $\mathbf{W}$ must be positive definite; but when *asynchronous* updates are used (one node at a time) it is sufficient that the diagonal elements of $\mathbf{W}$ be nonnegative; that is $w_{ii} \geq 0$. These conditions are more stringent than those for the continuous-time network.



17. Discrete-time Hopfield network.

Hopfield Associative Memory

One of Hopfield's original applications was the associative memory. An associative memory is a device which accepts an input pattern and produces as an output the *stored pattern* which is most closely associated with the input. In the Hopfield associative memory, the input/output patterns are binary. For example, the input pattern may be a noisy version of one of the stored patterns. The function of the associative memory is to recall the corresponding stored pattern, producing a clean version of the pattern at the output. In the Hopfield network the stored patterns are encoded in the weights of the network.

When used as an associative memory the input/output patterns are represented in bipolar form; that is, with 1s and -1s. Thus, the activation function in Eq. 1 is modified to yield a *bipolar* output.

The simplest approach for programming the weights of the Hopfield associative memory is to use the outer product method [53]. If we let $\boldsymbol{z}_i$, i=1,2,...,M represent the M (bipolar) patterns that are to be stored in the network, then the weight matrix is programmed as follows:

$$\boldsymbol{W} = \sum_{i=1}^{M} \boldsymbol{z}_i \boldsymbol{z}_i^T - \alpha M \boldsymbol{I} \tag{49}$$

where $0 \leq \alpha \leq 1$. Note first that W is symmetric. Note also that each term in the summation contributes a value of +1 to the diagonal elements of W, so that the total contribution from the summation is M. The second term serves only to decrease the value of the diagonal elements by a fraction α. With α=0 the diagonal elements of W equal M, and with α=1 they equal 0. The effect of having zero verses nonzero diagonal elements in W is discussed in [18]. The storage and recall properties of this method are well known. It has been shown that if the input patterns are less than N/2 away in Hamming distance from a stored pattern, and if the M stored patterns are chosen at random, the maximum asymptotic value of M for which all M of the patterns can be perfectly recalled is [71, 87]:

$$M \leq \frac{N}{4 \ln N} \tag{50}$$

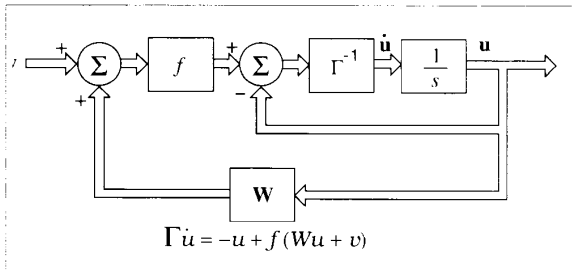
For example, if $N = 256$ then $M \leq 12$. Alternatively, if we allow a small fraction of the bits in the recalled pattern to be in error, then the capacity is $M \leq 0.138\,N$ [51]. With M=0.138 N, approximately 1.6 percent of the bits in the recalled pattern are in error. For M greater than 0.138 N, the number of erroneous bits increases rapidly (an avalanche effect), rendering the network useless as an associative memory. These results assume that the weight matrix is formed using the outer product design method in Eq. 49. Other design techniques which provide improved storage capacity can be found in [31, 136]. The capacities achieved with these methods are closer to $M = N$, which is much better than those achieved with the outer product method, and are in fact the maximum possible for the Hopfield network [1].

Continuous-Time Recurrent Neural Network

The next model that we discuss is very similar to the Hopfield network. This model is called the Continuous-Time Recurrent Neural Network (CTRNN), and is described in [109]. Like the Hopfield network, this network consists of a single layer of nodes which are fully interconnected. Generally, one subset of the nodes serve as the "input nodes," while another serves as the "output nodes." These subsets are denoted A and Ω, respectively. The dynamics of the network are described by the following differential equation:

$$\tau_i \dot{u}_i(t) = -u_i(t) + f\left(\sum_{j=1}^{N} w_{i,j}\, u_j(t) + v_i\right) \quad i = 1,2,\ldots,N \tag{51}$$

where $u_i(t)$ is the state of the ith node, $w_{i,j}$ is the weight which connects node j to node i, v_i is the input to the ith node, and $f(\cdot)$ is the nonlinear activiation function (typically a sigmoid). The dynamics of this system are shown in Fig. 18. It is instructive to compare this diagram with that of the Hopfield network in Fig. 16; the two models are closely related. In fact, one can show that the Hopfield Network is related the CTRNN by a simple affine transformation. If we let $\boldsymbol{y}_H(t) = \boldsymbol{W}\boldsymbol{u}_R(t) + \boldsymbol{v}$, where $\boldsymbol{y}_H(t)$ is the state vector of the Hopfield network and $\boldsymbol{u}_R(t)$ is the state vector of the recurrent neural net, then the defining equations in Fig. 18 can easily be obtained from the equations accompanying Fig. 16. (This transformation requires that $\mathbf{W}$ be invertible.) Training algorithms for the CTRNN are discussed in [107, 109].



18. Continuous-time recurrent neural network.

Discrete-Time Recurrent Neural Network

A discrete-time approximation to the CTRNN is the discrete-time recurrent neural network (DTRNN) [117, 150]. Consider the following discrete-time approximation of Eq. 51:

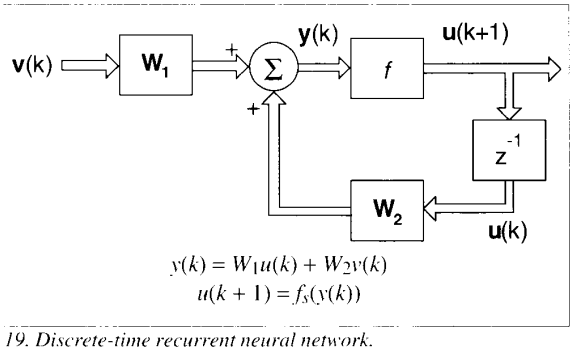
$$u_i(k+1) = f\left(\sum_{j=1}^{N} w_{i,j}\, u_j(k)\right) + v_i(k)) \tag{52}$$

Now if we allow each node to weight not only the outputs from other nodes, but also the components of the input vector, then we can re-express the above equation as:

$$u_i(k+1) = f\left(\sum_{j=0}^{N+M} w_{i,j}\, u_j(k)\right) \tag{53}$$

where $u_i(k)$ is reexpressed as:

$$u_i(k) = \begin{cases} 1 & i = 0 \\ u_i(k) & i = 1,2,\ldots,N \\ v_{i-N}(k) & i = N+1,\ldots,N+M \end{cases} \tag{54}$$



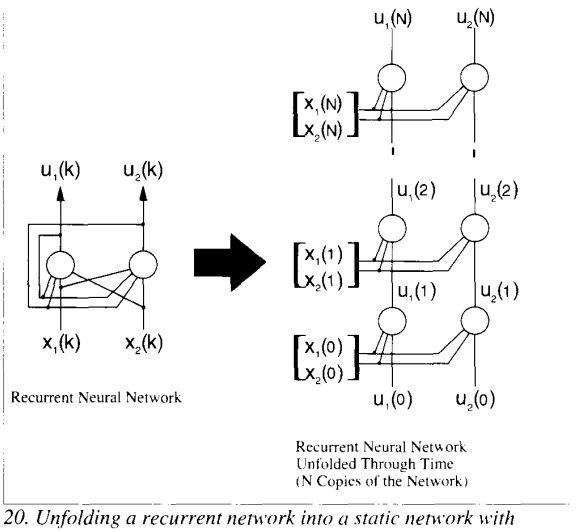
19. Discrete-time recurrent neural network.

N is the number of nodes in the network, and M the size of the input vector $\mathbf{v}$. We define $u_0(k)$=1 to account for the bias weights, $w_{i,0}$, just as we did in the MLP. Equation 53 then describes the node equations for the DTRNN. The system diagram for this network is shown in Fig. 19.

DTRNN Functional Capabilities

The behavior of the DTRNN is similar to the CTRNN discussed above. However, the DTRNN can take on a unique interpretation based on its discrete-time nature. Recall that each node in the network is capable of implementing simple logic functions. This property, along with the feedback connections and the discrete time delays, make it possible for the DTRNN to emulate Deterministic Finite Automata (DFAs) [4, 88]. With this interpretation, it is easy to see how the DTRNN can perform such tasks as sequence recognition.

It is also easy to show that the DTRNN can be reduced to an ordinary MLP. First we note that when the connection matrix $\boldsymbol{W}_2$ is lower triangular, all weight connections are feed



20. Unfolding a recurrent network into a static network with shared weights.

forward. Then with some additional constraints we can form a weight matrix which imposes a layered structure onto the nodes. The resulting system is identical to an MLP except that it takes n time steps to feed a pattern from the input to the output layer of an n layer network.

DTRNN Learning Algorithms

One way to learn the weights in a Discrete-Time Recurrent Neural Network is to convert the network from a feedback system into a purely feedforward system by *unfolding the network over time*. The idea is that if the system processes a signal that is n time steps long, then we create n copies of the network. The feedback connections are modified so that they are now feedforward connections from one network to the subsequent network (Fig. 20).

The network can then be trained as if it is one giant feedforward network with the copied weights being treated as shared weights. This approach to learning is called *Backpropagation Through Time* [119]. Another approach, called *Truncated Backpropagation Through Time*, tries to approximate the true gradient by only unfolding the network over the last m time steps [151]. In this case, only m copies of the network are made, and normal Backpropagation with weight sharing is used. For some problems performance may be sacrificed if critical information occurs more than m time steps in the past. The advantage of these learning algorithms is that the computation is simplified, since normal Backpropagation with weight sharing can be used. However, there is a large memory cost to maintain several copies of the network.

Another approach is to calculate the gradient recursively. This approach is commonly known as *Real Time Recurrent Learning* (RTRL) [150]. The advantage of this approach is that it is more memory efficient than Backpropagation Through Time, although there is a computational cost incurred by the recursive process. Here again, the learning algorithm is derived using a gradient search to minimize a Sum of Squared Error criterion. The criterion function is given by:

$$J(\boldsymbol{w}) = \sum_{p=1}^{P} J_p(\boldsymbol{w}) \tag{55}$$

where P is the number of training sequences, and $J_p(\boldsymbol{w})$ is the total squared error for the pth sequence:

$$J_p(\boldsymbol{w}) = \frac{1}{2} \sum_{k=1}^{K_p} \sum_{j \in \Omega} (d_j(k) - u_j(k))^2 \tag{56}$$

In this equation, K_p is the length of the pth training sequence. Recall also that Ω represents the set of output nodes in the network. Applying the gradient operator to Eq. 55 and substituting into the weight update formula yields:

$$w_{j,i}(m+1) = w_{j,i}(m) - \mu \sum_{p=1}^{P} \frac{\partial J_p(\boldsymbol{w})}{\partial w_{j,i}}\bigg|_{\boldsymbol{w}(m)} \tag{57}$$

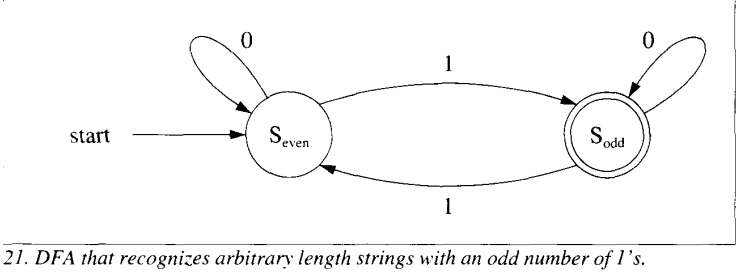
where m is the iteration index of the weights. Note that m is necessarily different from k. The later is an index on the time scale of the input sequences, while the former is an index on the time scale of weight updates which is much slower. The partial derivative in Eq. 57 is of the form [150]:

$$\frac{\partial J_p(\boldsymbol{w})}{\partial w_{j,i}} = -\sum_{k=1}^{K_p} \sum_{h \in \Omega} (d_h(k) - u_h(k)) p_{j,i}^{h}(k) \tag{58}$$

where $p_{j,i}^{h}(k)$ is a partial derivative defined as:

$$p_{j,i}^{h}(k) = \frac{\partial u_h(k)}{\partial w_{j,i}} \tag{59}$$

and can be expressed as:



21. DFA that recognizes arbitrary length strings with an odd number of 1's.

$$p_{j,i}^{h}(k) = f'\Big(\sum_{\alpha=0}^{N+M} w_{h,\alpha}\, u_\alpha(k-1)\Big) \cdot \Big[\delta_{h,j}\, u_i(k-1) + \sum_{\beta=1}^{N} w_{h,\beta} \frac{\partial u_\beta(k-1)}{\partial w_{j,i}}\Big] \tag{60}$$

$$= f'\Big(\sum_{\alpha=0}^{N+M} w_{h,\alpha}\, u_\alpha(k-1)\Big) \cdot \Big[\delta_{h,j}\, u_i(k-1) + \sum_{\beta=1}^{N} w_{h,\beta}\; p_{j,i}^{\beta}(k-1)\Big]$$

where $\delta_{h,j}$ is the Kronecker delta function. Note that in an n node network, we must maintain n terms for each weight. Since there are n^2 weights, there are n^3 of these terms. In addition, the calculation of each of these terms requires a summation over n terms (the sum over β). Thus, each time iteration requires $O(n^4)$ calculations.

Finally, substituting Eq. 58 into equation 57 gives:

$$w_{j,i}(m+1) = w_{j,i}(m) + \mu \sum_{p=1}^{P} \sum_{k=1}^{K_p} \sum_{h \in \Omega} (d_h(k) - u_h(k))\, p_{j,i}^{h}(k) \tag{61}$$

Note in this equation that the weights are updated only once each time through the training set. However, it may be desirable to perform updates each time a sequence is presented to the network.

Recall that in the derivation of the Backpropagation algorithm the true gradient was approximated by an instantaneous estimate based on a single sample. Using a similar approach here, one obtains the following approximation:

$$w_{j,i}(m+1) = w_{j,i}(m) - \mu \left.\frac{\partial J_{m \bmod P}(\mathbf{w})}{\partial w_{j,i}}\right|_{\mathbf{w}(m)} \tag{62}$$

The learning algorithm with this approximation is summarized in Table 5.

An example

Logic problems were among the first to be considered for learning in static networks because of their solid theoretical foundation. It seems natural then that Deterministic Finite Automata (DFAs) (logic gates coupled with a finite memory) be among the first problems investigated in recurrent neural networks. We have seen that Multilayer Perceptrons are capable of implementing arbitrary logic functions if we allow a sufficient number of hidden layer nodes. Similarly, it can be shown that the DTRNN is capable of implementing arbitrary DFAs [4]. Of course, we are interested primarily in *learning* DFAs from a set of examples.

Since the XOR problem has a great deal of historical significance, it is natural to look at its time-dependent counterpart— the *parity problem*. The DFA for parity is shown in Fig. 21. The circles in the figure correspond to the *states* of the DFA. There are two states, S_{even} and S_{odd}. The directed arcs between the states are *transitions* indicating how the states change as the input string is processed. We start in state S_{even} and process the input string one character at a time, making transitions between the states. For this DFA, the state labeled S_{even} corresponds to the case when the number of '1's in any prefix of the input string is an even number, while the state labeled S_{odd} corresponds to an odd number of '1's. The double circle on state S_{odd} indicates that if we end in this state after processing an input string, then that string is *accepted* by the DFA; otherwise the string is *rejected*. Thus, this DFA accepts all strings which have an odd number of 1's and rejects all other strings.

The problem of learning, or *inferring* DFAs is typically stated as follows: Given a set of positive and negative example strings, find a DFA which accepts the positive strings and rejects the negative strings. It turns out that such problems are relatively easy [35] unless we limit the number of states that the DFA can have, in which case the problem is NP-complete [6, 41].

A DTRNN with 3 nodes, one of which was chosen as the output node, was trained to solve the parity problem. The training set consisted of a set of 1000 strings randomly sampled from the set of all strings of length up to and including length five. Thus, many of these strings were presented repeatedly to the network. We found it advantageous to repeat the shorter strings more frequently than the longer strings. In our case each string appeared with a relative frequency of occurrence equal to:

$$p_i = \frac{\left(\frac{1}{2}\right)^{l_i}}{\sum_j \left(\frac{1}{2}\right)^{l_j}} \tag{63}$$

where l_i is the length of the ith string. A popular alternative approach is to train on shorter strings first, and then add longer strings in the later stages of learning [82]. The network found a solution after 91 training epochs. To determine the DFA learned by the network, we replaced the sigmoids with hard limiting nonlinearities and extracted the logic functions implemented by the network. The resulting DFA is shown in Fig. 22. While this DFA looks considerably more complex than the DFA shown in Fig. 21, it can easily be shown that the two DFAs are equivalent.

The process of *extracting* the DFA from a recurrent neural network is not, in general, as easy as replacing sigmoids by hard limiters [38, 141]. Often the network utilizes the transition region of the sigmoid to encode the states of the DFA which can greatly complicate matters. Some researchers have found that the representations of the states in the DFA can be distributed throughout the state space in a chaotic manner,

node outputs are updated asynchronously using the equations in Box 1. Hopfield [19] also demonstrated that the net converges when graded nonlinearities similar to the sigmoid nonlinearity in Fig. 1 are used. When the Hopfield net is used as an associative memory, the net output after convergence is used directly as the complete restored memory. When the Hopfield net is used as a classifier, the output after convergence must be compared to the M exemplars to determine if it matches an exemplar exactly. If it does, the output is that class whose exemplar matched the output pattern. If it does not then a "no match" result occurs.

Box 1. Hopfield Net Algorithm

Step 1. Assign Connection Weights

$$t_{ij} = \begin{cases} \sum_{s=0}^{M-1} x_i^s x_j^s, & i \neq j \\ 0, & i = j, \ 0 \leq i, j \leq M-1 \end{cases}$$

In this Formula t_{ij} is the connection weight from node i to node j and x_i^s which can be +1 or −1 is element i of the exemplar for class s.

Step 2. Initialize with Unknown Input Pattern

$$\mu_i(0) = x_i, \quad 0 \leq i \leq N-1$$

In this Formula $\mu_i(t)$ is the output of node i at time t and x_i which can be +1 or −1 is element i of the input pattern.

Step 3. Iterate Until Convergence

$$\mu_j(t+1) = f_h\left[\sum_{i=0}^{N-1} t_{ij}\,\mu_i(t)\right], \quad 0 \leq j \leq M-1$$

The function f_h is the hard limiting nonlinearity from Fig. 1. The process is repeated until node outputs remain unchanged with further iterations. The node outputs then represent the exemplar pattern that best matches the unknown input.

Step 4. Repeat by Going to Step 2

The behavior of the Hopfield net is illustrated in Fig. 5. A Hopfield net with 120 nodes and thus 14,400 weights was trained to recall the eight exemplar patterns shown at the top of Fig. 5. These digit-like black and white patterns contain 120 pixels each and were hand crafted to provide good performance. Input elements to the net take on the value +1 for black pixels and −1 for white pixels. In the example presented, the pattern for the digit "3" was corrupted by randomly reversing each bit independently from +1 to −1 and vice versa with a probability of 0.25. This pattern was then applied to the net at time zero. Patterns produced at the output of the net on iterations zero to seven are presented at the bottom of Fig. 5. The corrupted input pattern is present unaltered at iteration zero. As the net iterates the output becomes more and more like the correct exemplar pattern until at iteration six the net has converged to the pattern for the digit three.

The Hopfield net has two major limitations when used as a content addressable memory. First, the number of patterns that can be stored and accurately recalled is severely limited. If too many patterns are stored, the net may converge to a novel spurious pattern different from all exemplar patterns. Such a spurious pattern will produce a "no match" output when the net is used as a classifier. Hopfield [18] showed that this occurs infrequently when exemplar patterns are generated randomly and the number of classes (M) is less than .15 times the number of input elements or nodes in the net (N). The number of classes is thus typically kept well below .15N. For example, a Hopfield net for only 10 classes might require more than 70 nodes and more than roughly 5,000 connection weights. A second limitation of the Hopfield net is that an exemplar pattern will be unstable if it shares many bits in common with another exemplar pattern. Here an exemplar is considered unstable if it is applied at time zero and the net converges to some other exemplar. This problem can be eliminated and performance can be improved by a number of orthogonalization procedures [14, 46].

THE HAMMING NET

The Hopfield net is often tested on problems where inputs are generated by selecting an exemplar and reversing bit values randomly and independently with a given probability [18, 12, 46]. This is a classic problem in communications theory that occurs when binary fixed-length signals are sent through a memoryless binary symmetric channel. The optimum minimum error classifier in this case calculates the Hamming distance to the exemplar for each class and selects that class with the minimum Hamming distance [10]. The Hamming distance is the number of bits in the input which do not match the corresponding exemplar bits. A net which will be called a Hamming net implements this algorithm using neural net components and is shown in Fig. 6.

The operation of the Hamming net is described in Box 2. Weights and thresholds are first set in the lower subnet such that the matching scores generated by the outputs of the middle nodes of Fig. 6 are equal to N minus the Hamming distances to the exemplar patterns. These matching scores will range from 0 to the number of elements in the input (N) and are highest for those nodes corresponding to classes with exemplars that best match the input. Thresholds and weights in the MAXNET subnet are fixed. All thresholds are set to zero and weights from each node to itself are 1. Weights between nodes are inhibitory with a value of $-\epsilon$ where $\epsilon < 1/M$.

After weights and thresholds have been set, a binary pattern with N elements is presented at the bottom of the Hamming net. It must be presented long enough to allow

the matching score outputs of the lower subnet to settle and initialize the output values of the MAXNET. The input is then removed and the MAXNET iterates until the output of only one node is positive. Classification is then complete and the selected class is that corresponding to the node with a positive output.

The behavior of the Hamming net is illustrated in Fig. 7.

Box 2. Hamming Net Algorithm

Step 1. Assign Connection Weights and Offsets

In the lower subnet:

$$w_{ij} = \frac{x_i^j}{2}, \qquad \theta_j = \frac{N}{2},$$
$$0 \le i \le N-1, \qquad 0 \le j \le M-1$$

In the upper subnet:

$$t_{kl} = \begin{cases} 1, & k = l \\ -\varepsilon, & k \ne l, \end{cases} \qquad \varepsilon < \frac{1}{M},$$
$$0 \le k, l \le M-1$$

In these equations w_{ij} is the connection weight from input i to node j in the lower subnet and θ is the threshold in that node. The connection weight from node k to node l in the upper subnet is t_{kl} and all thresholds in this subnet are zero. x_i^j is element i of exemplar j as in Box 1.

Step 2. Initialize with Unknown Input Pattern

$$\mu_j(0) = f_t\left(\sum_{i=0}^{N-1} w_{ij}x_i - \theta_j\right)$$
$$0 \le j \le M-1$$

In this equation $\mu_j(t)$ is the output of node j in the upper subnet at time t, x_i is element i of the input as in Box 1, and f_t is the threshold logic nonlinearity from Fig. 1. Here and below it is assumed that the maximum input to this nonlinearity never causes the output to saturate.

Step 3. Iterate Until Convergence

$$\mu_j(t+1) = f_t\left(\mu_j(t) - \varepsilon\sum_{k \ne j}\mu_k(t)\right)$$
$$0 \le j, k \le M-1$$

This process is repeated until convergence after which the output of only one node remains positive.

Step 4. Repeat by Going to Step 2

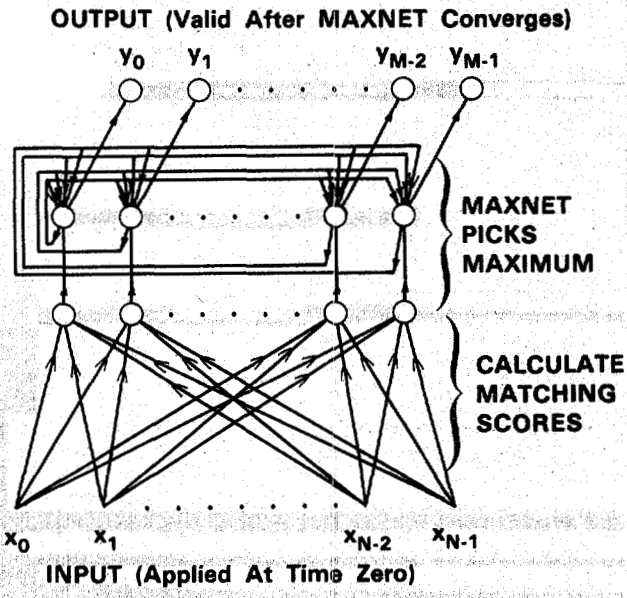


Figure 6. A feed-forward Hamming net maximum likelihood classifier for binary inputs corrupted by noise. The lower subnet calculates N minus the Hamming distance to M exemplar patterns. The upper net selects that node with the maximum output. All nodes use threshold-logic nonlinearities where it is assumed that the outputs of these nonlinearities never saturate.

The four plots in this figure show the outputs of nodes in a MAXNET with 100 nodes on iterations 0, 3, 6, and 9. These simulations were obtained using randomly selected exemplar patterns with 1000 elements each. The exemplar for class 50 was presented at time zero and then removed. The matching score at time zero is maximum (1000) for node 50 and has a random value near 500 for other nodes. After only 3 iterations, the outputs of all nodes except node 50 have been greatly reduced and after 9 iterations only the output for node 50 is non-zero. Simulations with different probabilities of reversing bits on input patterns and with different numbers of classes and elements in the input patterns have demonstrated that the MAXNET typically converges in less than 10 iterations in this application [25]. In addition, it can be proven that the MAXNET will always converge and find the node with the maximum value when $\varepsilon < 1/M$ [25].

The Hamming net has a number of obvious advantages over the Hopfield net. It implements the optimum minimum error classifier when bit errors are random and independent, and thus the performance of the Hopfield net must either be worse than or equivalent to that of the Hamming net in such situations. Comparisons between the two nets on problems such as character recognition, recognition of random patterns, and bibliographic retrieval have demonstrated this difference in performance [25]. The Hamming net also requires many fewer connections than the Hopfield net. For example, with 100 inputs and 10 classes the Hamming net requires

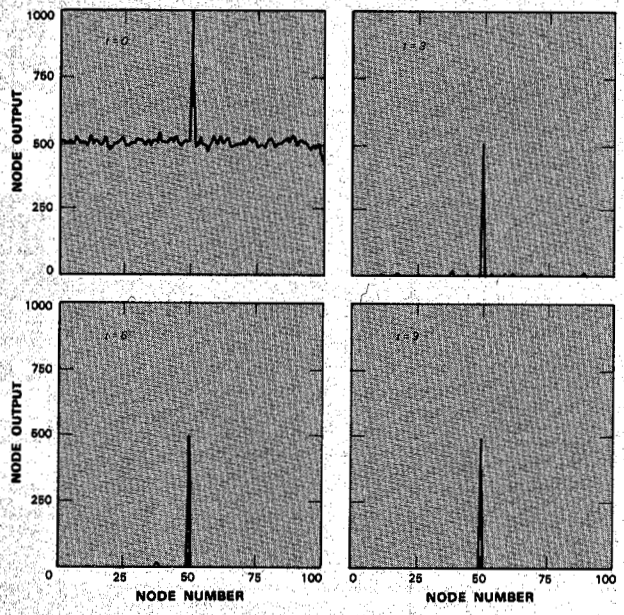


Figure 7. Node outputs for a Hamming net with 1,000 binary inputs and 100 output nodes or classes. Output values of all 100 nodes are presented at time zero and after 3, 6, and 9 iterations. The input was the exemplar pattern corresponding to output node 50.

only 1,100 connections while the Hopfield net requires almost 10,000. Furthermore, the difference in number of connections required increases as the number of inputs increases, because the number of connections in the Hopfield net grows as the square of the number of inputs while the number of connections in the Hamming net grows linearly. The Hamming net can also be modified to be a minimum error classifier when errors are generated by reversing input elements from +1 to −1 and from −1 to +1 asymmetrically with different probabilities [25] and when the values of specific input elements are unknown [2]. Finally, the Hamming net does not suffer from spurious output patterns which can produce a "no-match" result.

SELECTING OR ENHANCING THE MAXIMUM INPUT

The need to select or enhance the input with a maximum value occurs frequently in classification problems. Several different neural nets can perform this operation. The MAXNET described above uses heavy lateral inhibition similar to that used in other net designs where a maximum was desired [20, 22, 9]. These designs create a "winner-take-all" type of net whose design mimics the heavy use of lateral inhibition evident in the biological neural nets of the human brain [21]. Other techniques to pick a maximum are also possible [25]. One is illustrated in Fig. 8. This figure shows a comparator subnet which is described in [29]. It uses threshold logic nodes to pick the maximum of two inputs and then feeds this maximum value forward. This net is useful when the maximum value must be passed unaltered to the output. Comparator subnets can be layered into roughly $\log_2(M)$ layers to pick the maximum of M inputs. A net that uses these subnets to pick the maximum of 8 inputs is presented in Fig. 9.

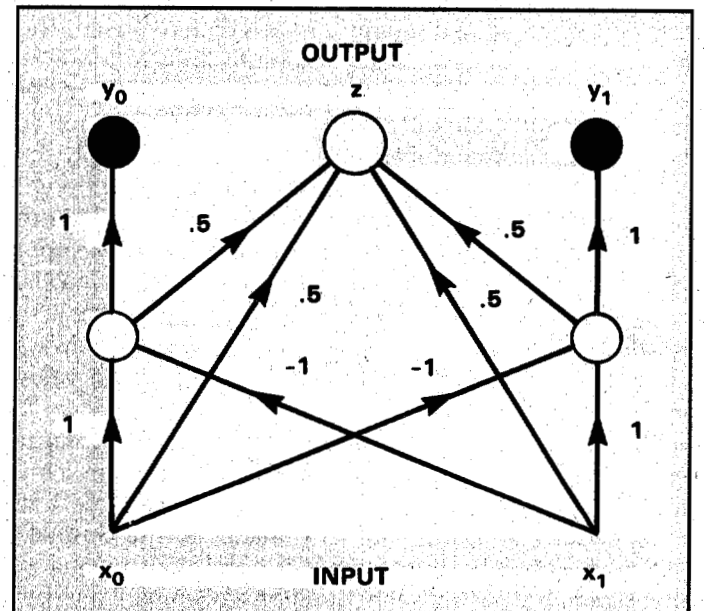


Figure 8. A comparator subnet that selects the maximum of two analog inputs. The output labeled z is the maximum value and the outputs labeled y_0 and y_1 indicate which input was maximum. Internal thresholds on threshold logic nodes (open circles) and hard limiting nodes (filled circles) are zero. Weights are as shown.

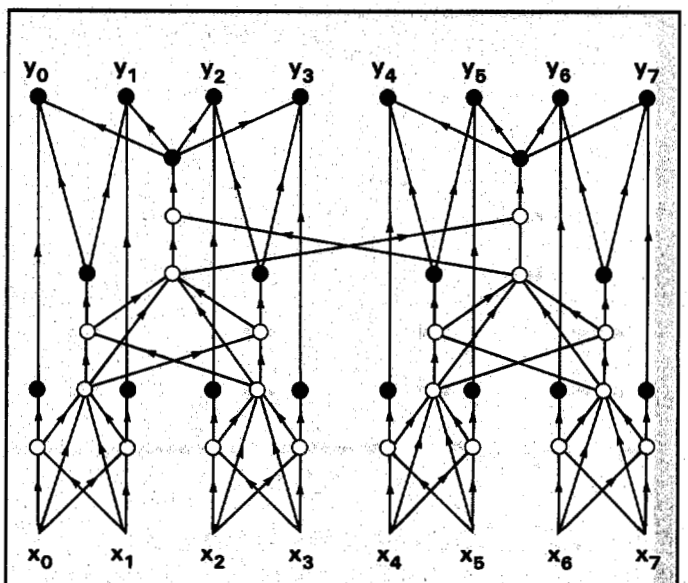


Figure 9. A feed-forward net that determines which of eight inputs is maximum using a binary tree and comparator subnets from Fig. 8. After an input vector is applied, only that output corresponding to the maximum input element will be high. Internal thresholds on threshold-logic nodes (open circles) and on hard limiting nodes (filled circles) are zero except for the output nodes. Thresholds in the output nodes are 2.5. Weights for the comparator subnets are as in Fig. 8 and all other weights are 1.