

- (a) The sum of the eigenvalues of the matrix $\mathbf{W}$ is zero. Why?
 (b) The state space of the network is a subspace of $\mathbb{R}^5$. Specify the configuration of this subspace.
 (c) Specify the subspace $\mathcal{M}$ spanned by the fundamental memory vectors, and the null subspace $\mathcal{N}$ of the weight matrix $\mathbf{W}$. What are the fixed points (stable states) and spurious states of the network?

(The reader may wish to refer to the paper by deSilva and Attikiouzel (1992) for a more detailed description of the dynamics of the network described here.)

- 14.11 Figure P14.11 shows a piecewise-linear form of nonmonotonic activation function. The recalling dynamics of the Hopfield network using this approximation are defined by

$$\frac{d}{dt}\mathbf{v}(t) = -\mathbf{v}(t) + \mathbf{W}\mathbf{x}(t), \quad \mathbf{x}(t) = \text{sgn}(\mathbf{v}(t)) - k\mathbf{v}(t)$$

where $\mathbf{v}(t)$ is the vector of induced local fields, $\mathbf{W}$ is the synaptic weight matrix, $\mathbf{x}(t)$ is the state (output) vector, and $-k$ is a negative constant slope. Let $\mathbf{v}$ be an equilibrium state of the network that lies in the quadrant of the fundamental memory ξ_1 , and let

$$\bar{\mathbf{x}} = \text{sgn}(\mathbf{v}) - k\mathbf{v}$$

Show that $\bar{\mathbf{x}}$ is characterized by the following three conditions (Yoshizawa et al., 1993):

- (a) $\sum_{i=1}^N x_i \xi_{\mu,i} = 0, \quad \mu = 2, 3, \dots, M$
 (b) $\sum_{i=1}^N x_i \xi_{1,i} = M$
 (c) $x_i < 1, \quad i = 1, 2, \dots, N$

where $\xi_1, \xi_2, \dots, \xi_M$ are the fundamental memories stored in the network, $\xi_{\mu,i}$ is the i th element of ξ_{μ} , x_i is the i th element of $\bar{\mathbf{x}}$, and N is the number of neurons.

- 14.12 Consider the simple neurodynamical model described by the system of equations

$$\frac{dv_j}{dt} = -v_j + \sum_i w_{ji} \varphi(v_i) + I_j, \quad j = 1, 2, \dots, N$$

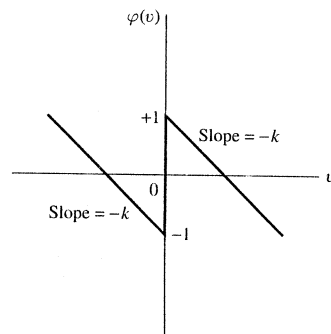


FIGURE P14.11

The system described always converges to a unique point attractor, provided that the synaptic weights w_{ji} satisfy the condition

$$\sum_j \sum_i w_{ji}^2 < \frac{1}{(\max |\varphi'|)^2}$$

where $\varphi' = d\varphi/dv$. Explore the validity of this condition. You may refer to the paper (Atiya, 1987) where this condition is derived.

Cohen-Grossberg Theorem

- 14.13 Consider the Lyapunov function E defined in Eq. (14.57). Show that

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

provided that the conditions in Eqs. (14.59) to (14.61) are satisfied.

- 14.14 In Section 14.10 we derived the Lyapunov function of the BSB model by applying the Cohen-Grossberg theorem. In carrying out the derivation, we omitted some of the details leading to Eq. (14.73). Fill in the details.

- 14.15 Figure P14.15 shows a plot of the nonmonotonic activation function due to Morita (1993) discussed in note 6. This activation function is used in place of the hyperbolic tangent function in the construction of a Hopfield network. Is the Cohen-Grossberg theorem applicable to the associative memory so constructed? Justify your answer.

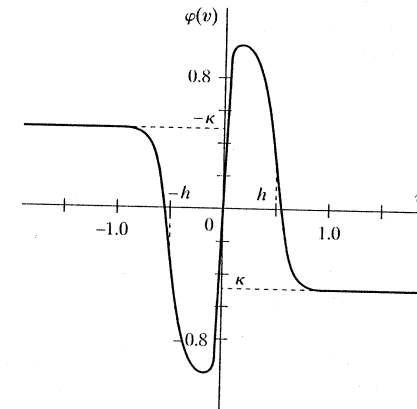


FIGURE P14.15

Dynamically Driven Recurrent Networks

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 architecture.

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

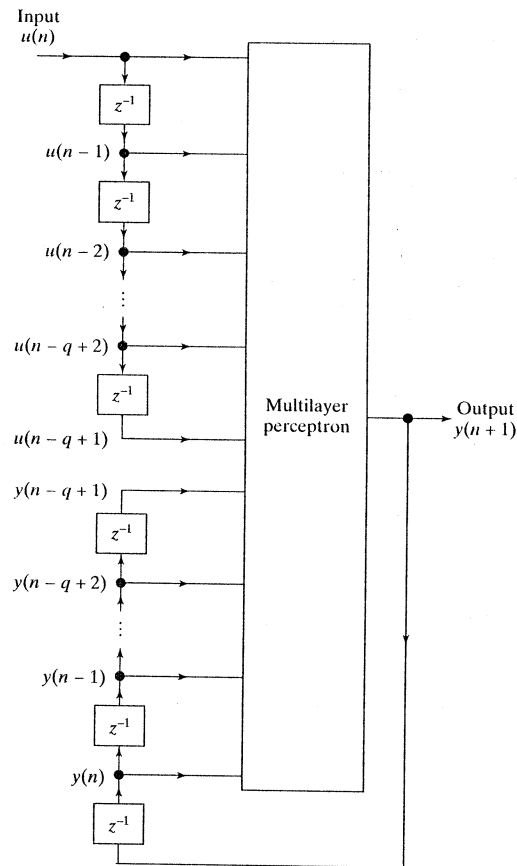


FIGURE 15.1 Nonlinear autoregressive with exogenous inputs (NARX) model.

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 $u(n), u(n-1), \dots, u(n-q+1)$, which represent *exogenous* inputs originating from outside the network.
- Delayed values of the output, namely, $y(n), y(n-1), \dots, y(n-q+1)$, 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

$$y(n+1) = F(y(n), \dots, y(n-q+1), u(n), \dots, u(n-q+1)) \quad (15.1)$$

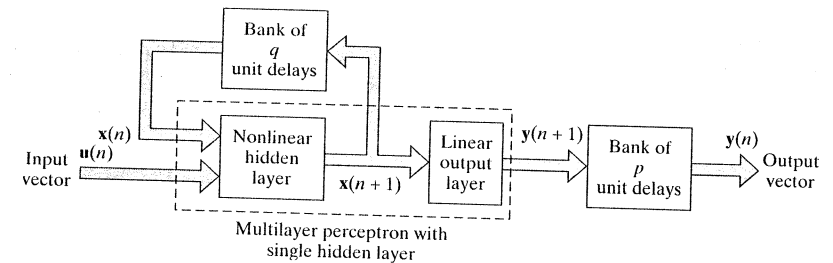


FIGURE 15.2 State-space model.

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 *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 delay 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-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:

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

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

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

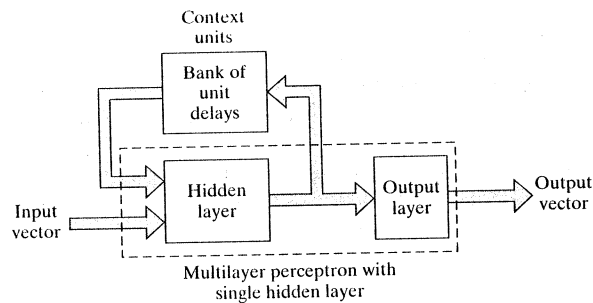


FIGURE 15.3 Simple Recurrent network (SRN).

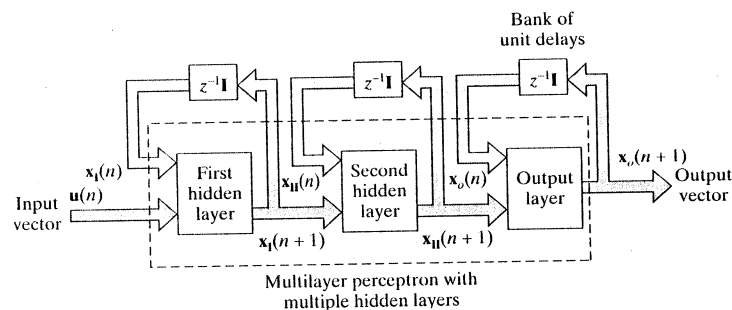


FIGURE 15.4 Recurrent multilayer perceptron.

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 $\mathbf{x}_1(n)$ denote the output of the first hidden layer, $\mathbf{x}_{11}(n)$ 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:

$$\begin{aligned} \mathbf{x}_1(n+1) &= \varphi_1(\mathbf{x}_1(n), \mathbf{u}(n)) \\ \mathbf{x}_{11}(n+1) &= \varphi_{11}(\mathbf{x}_{11}(n), \mathbf{x}_1(n+1)) \\ &\vdots \\ \mathbf{x}_o(n+1) &= \varphi_o(\mathbf{x}_o(n), \mathbf{x}_k(n+1)) \end{aligned} \quad (15.4)$$

where $\varphi_1(\cdot, \cdot), \varphi_{11}(\cdot, \cdot), \dots, \varphi_o(\cdot, \cdot)$ 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

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

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

$$v_k = \sum_i \sum_j w_{kij} x_i u_j \quad (15.6)$$

we refer to the neuron as a *second-order neuron*. The second-order neuron k uses a single weight, w_{kij} , 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:

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

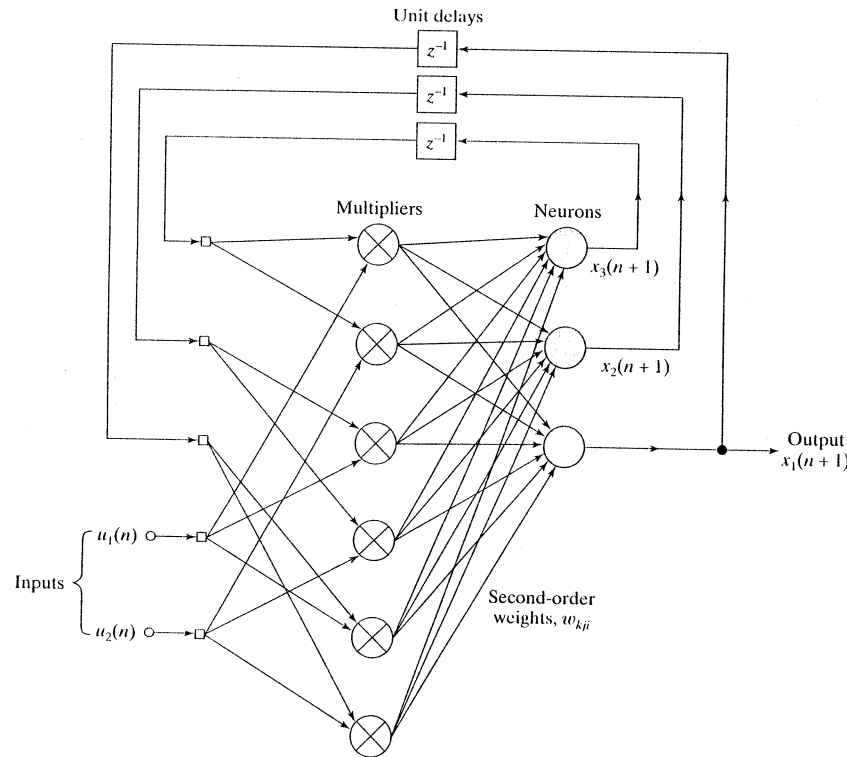


FIGURE 15.5 Second-order recurrent network; bias connections to the neurons are omitted to simplify the presentation. The network has 2 inputs and 3 state neurons, hence the need for $3 \times 2 = 6$ multipliers.

and

$$x_k(n+1) = \varphi(v_k(n)) = \frac{1}{1 + \exp(-v_k(n))} \quad (15.8)$$

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 $x_j(n)u_i(n)$ 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

$$\delta(x_i, u_j) = x_k \quad (15.9)$$

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):

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

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

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 $\varphi: \mathbb{R}^q \rightarrow \mathbb{R}^q$ is a diagonal map described by

$$\varphi: \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_q \end{bmatrix} \rightarrow \begin{bmatrix} \varphi(x_1) \\ \varphi(x_2) \\ \vdots \\ \varphi(x_q) \end{bmatrix} \quad (15.12)$$

for some memoryless, component-wise nonlinearity $\varphi: \mathbb{R} \rightarrow \mathbb{R}$. 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.

For the interpretation of matrices $\mathbf{W}_a$, $\mathbf{W}_b$, and $\mathbf{C}$, and nonlinear function $\varphi(\cdot)$, we may say:

- The matrix $\mathbf{W}_a$ represents the synaptic weights of the q neurons in the hidden layer that are connected to the feedback nodes in the input layer. The matrix $\mathbf{W}_b$ represents the synaptic weights of these hidden neurons that are connected to the source nodes in the input layer. It is assumed that the bias terms for the hidden neurons are absorbed in the weight matrix $\mathbf{W}_b$.
- The matrix $\mathbf{C}$ represents the synaptic weights of the p linear neurons in the output layer that are connected to the hidden neurons. It is assumed that the bias terms for the output neurons are absorbed in the weight matrix $\mathbf{C}$.
- The nonlinear function $\varphi(\cdot)$ represents the sigmoid activation function of a hidden neuron. The activation function typically takes the form of a hyperbolic tangent function:

$$\varphi(x) = \tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} \quad (15.13)$$

or a logistic function:

$$\varphi(x) = \frac{1}{1 + e^{-x}} \quad (15.14)$$

An important property of a recurrent network described by the state-space model of Eqs. (15.10) and (15.11) is that it can *approximate* a wide class of nonlinear dynamical systems. However, the approximations are only valid on compact subsets of the state space and for finite time intervals, so that interesting dynamical characteristics are not reflected (Sontag, 1992).

Example 15.1

To illustrate the compositions of matrices $\mathbf{W}_a$, $\mathbf{W}_b$ and $\mathbf{C}$, consider the *fully connected recurrent network* shown in Fig. 15.6, where the feedback paths originate from the hidden neurons. In this example we have $m = 2$, $q = 3$, and $p = 1$. The matrices $\mathbf{W}_a$ and $\mathbf{W}_b$ are defined as follows:

$$\mathbf{W}_a = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix}$$

and

$$\mathbf{W}_b = \begin{bmatrix} b_1 & w_{14} & w_{15} \\ b_2 & w_{24} & w_{25} \\ b_3 & w_{34} & w_{35} \end{bmatrix}$$

where the first column of $\mathbf{W}_b$ consisting of b_1 , b_2 , and b_3 represents the bias terms applied to neurons 1, 2, and 3, respectively. The matrix $\mathbf{C}$ is a row vector defined by

$$\mathbf{C} = [1, 0, 0]$$

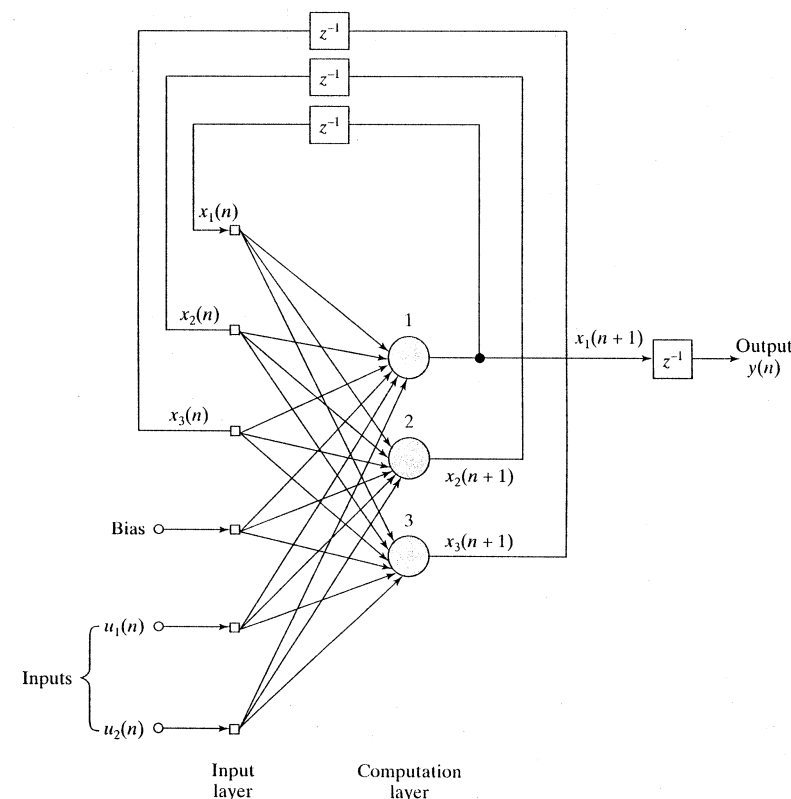


FIGURE 15.6 Fully connected recurrent network with 2 inputs, 2 hidden neurons, and 1 output neuron.

Controllability and Observability

In the study of system theory, stability, controllability, and observability are prominent features, each in its own fundamental way. In this section we discuss controllability and observability since they are usually treated together; stability is discussed in the previous chapter and will therefore not be pursued further.

As mentioned earlier, many recurrent networks can be represented by the state-space model shown in Fig. 15.2, where the state is defined by the output of the hidden layer fed back to the input layer via a set of unit delays. In that context, it is important to know whether or not the recurrent network is controllable and observable. Controllability is concerned with whether or not we can control the dynamic behavior of the recurrent network. Observability is concerned with whether or not we can observe the result of the control applied to the recurrent network. In that sense, observability is the dual of controllability.

A recurrent network is said to be *controllable* if an initial state is steerable to any desired state within a finite number of time steps; the output is irrelevant to this definition. The recurrent network is said to be *observable* if the state of the network can be determined from a finite set of input/output measurements. A rigorous treatment of controllability and observability of recurrent networks is beyond the scope of this book.⁵ Here we confine ourselves to *local* forms of controllability and observability, local in the sense that these notions apply in the neighborhood of an equilibrium state of the network (Levin and Narendra, 1993).

A state $\bar{\mathbf{x}}$ is said to be an *equilibrium state* of Eq. (15.10) if, for an input $\bar{\mathbf{u}}$, it satisfies the condition:

$$\bar{\mathbf{x}} = \varphi(\mathbf{A}\bar{\mathbf{x}} + \mathbf{B}\bar{\mathbf{u}}) \quad (15.15)$$

Without loss of generality, we may set $\bar{\mathbf{x}} = \mathbf{0}$ and $\bar{\mathbf{u}} = \mathbf{0}$. The equilibrium state is then described by

$$\mathbf{0} = \varphi(\mathbf{0})$$

In other words, the equilibrium point is represented by the origin $(\mathbf{0}, \mathbf{0})$.

Also without loss of generality, we may simplify the exposition by limiting ourselves to a single input, single output (SISO) system. We may then rewrite Eqs. (15.10) and (15.11) as follows, respectively:

$$\mathbf{x}(n+1) = \varphi(\mathbf{W}_a \mathbf{x}(n) + \mathbf{w}_b u(n)) \quad (15.16)$$

$$y(n) = \mathbf{c}^T \mathbf{x}(n) \quad (15.17)$$

where both $\mathbf{w}_b$ and $\mathbf{c}$ are q -by-1 vectors, $u(n)$ is the scalar input, and $y(n)$ is the scalar output. Since φ is continuously differentiable for the sigmoid function of Eq. (15.13) or that of Eq. (15.14), we may *linearize* Eq. (15.16) by expanding it as a Taylor series around the equilibrium point $\bar{\mathbf{x}} = \mathbf{0}$ and $\bar{\mathbf{u}} = \mathbf{0}$ and retaining first-order terms as follows:

$$\delta \mathbf{x}(n+1) = \varphi'(\mathbf{0}) \mathbf{W}_a \delta \mathbf{x}(n) + \varphi'(\mathbf{0}) \mathbf{w}_b \delta u(n) \quad (15.18)$$

where $\delta \mathbf{x}(n)$ and $\delta u(n)$ are small displacements applied to the state and input, respectively, and the q -by- q matrix $\varphi'(\mathbf{0})$ is the Jacobian of $\varphi(\mathbf{v})$ with respect to its argument $\mathbf{v}$, evaluated at $\mathbf{v} = \mathbf{0}$. We may thus describe the linearized system by writing

$$\delta \mathbf{x}(n+1) = \mathbf{A} \delta \mathbf{x}(n) + \mathbf{b} \delta u(n) \quad (15.19)$$

$$\delta y(n) = \mathbf{c}^T \delta \mathbf{x}(n) \quad (15.20)$$

where the q -by- q matrix $\mathbf{A}$ and the q -by-1 vector $\mathbf{b}$ are respectively defined by

$$\mathbf{A} = \varphi'(\mathbf{0}) \mathbf{W}_a \quad (15.21)$$

and

$$\mathbf{b} = \varphi'(\mathbf{0}) \mathbf{w}_b \quad (15.22)$$

The state equations (15.19) and (15.20) are in the standard linear form. We may therefore make use of well-known results on the controllability and observability of linear dynamical systems that are a standard part of mathematical control theory.

Local Controllability

From the linearized equation (15.19), we readily find that its repeated use yields the following equations:

$$\delta \mathbf{x}(n+1) = \mathbf{A} \delta \mathbf{x}(n) + \mathbf{b} \delta u(n)$$

$$\delta \mathbf{x}(n+2) = \mathbf{A} \delta \mathbf{x}(n+1) + \mathbf{b} \delta u(n+1)$$

$\vdots$

$$\delta \mathbf{x}(n+q) = \mathbf{A}^q \delta \mathbf{x}(n) + \mathbf{A}^{q-1} \mathbf{b} \delta u(n+q-1) + \cdots + \mathbf{A} \mathbf{b} \delta u(n+1) + \mathbf{b} \delta u(n)$$

where q is the dimensionality of the state space. Accordingly, we may state that (Levi and Narendra, 1993):

The linearized system represented by Eq. (15.19) is controllable if the matrix

$$\mathbf{M}_c = [\mathbf{A}^{q-1} \mathbf{b}, \dots, \mathbf{A} \mathbf{b}, \mathbf{b}] \quad (15.23)$$

is of rank q , that is, full rank, because then the linearized process equation (15.19) would have a unique solution.

The matrix $\mathbf{M}_c$ is called the *controllability matrix* of the linearized system.

Let the recurrent network described by Eqs. (15.16) and (15.17) be driven by sequence of inputs $\mathbf{u}_q(n)$ defined by

$$\mathbf{u}_q(n) = [u(n), u(n+1), \dots, u(n+q-1)]^T \quad (15.24)$$

Hence we may consider the mapping

$$\mathbf{G}(\mathbf{x}(n), \mathbf{u}_q(n)) = (\mathbf{x}(n), \mathbf{x}(n+q)) \quad (15.25)$$

where $\mathbf{G} : \mathbb{R}^{2q} \rightarrow \mathbb{R}^{2q}$. In Problem 15.4, it is shown that:

- The state $\mathbf{x}(n+q)$ is a nested nonlinear function of its past value $\mathbf{x}(n)$ and the inputs $u(n), u(n+1), \dots, u(n+q-1)$.
- The Jacobian of $\mathbf{x}(n+q)$ with respect to $\mathbf{u}_q(n)$, evaluated at the origin, is equal to the controllability matrix $\mathbf{M}_c$ of Eq. (15.23).

We may express the Jacobian of the mapping $\mathbf{G}$ with respect to $\mathbf{x}(n)$ and $\mathbf{u}_q(n)$, evaluated at the origin $(\mathbf{0}, \mathbf{0})$, as follows:

$$\begin{aligned} \mathbf{J}_{(\mathbf{0}, \mathbf{0})}^{(c)} &= \begin{bmatrix} \left(\frac{\partial \mathbf{x}(n)}{\partial \mathbf{x}(n)} \right)_{(\mathbf{0}, \mathbf{0})} & \left(\frac{\partial \mathbf{x}(n+q)}{\partial \mathbf{x}(n)} \right)_{(\mathbf{0}, \mathbf{0})} \\ \left(\frac{\partial \mathbf{x}(n)}{\partial \mathbf{u}_q(n)} \right)_{(\mathbf{0}, \mathbf{0})} & \left(\frac{\partial \mathbf{x}(n+q)}{\partial \mathbf{u}_q(n)} \right)_{(\mathbf{0}, \mathbf{0})} \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{I} & \mathbf{X} \\ \mathbf{0} & \mathbf{M}_c \end{bmatrix} \end{aligned} \quad (15.26)$$

where $\mathbf{I}$ is the identity matrix, $\mathbf{0}$ is the null matrix, and the entry $\mathbf{X}$ is of no interest. Because of its special form, the determinant of the Jacobian $\mathbf{J}_{(\mathbf{0}, \mathbf{0})}^{(c)}$ is equal to the product of the determinant of the identity matrix $\mathbf{I}$ (which equals 1) and the determinant of the controllability matrix $\mathbf{M}_c$. If $\mathbf{M}_c$ is of full rank, then so is $\mathbf{J}_{(\mathbf{0}, \mathbf{0})}^{(c)}$.

To proceed further, we need to invoke the *inverse function theorem*, which may be stated as follows (Vidyasagar, 1993):

Consider the mapping $\mathbf{f}: \mathbb{R}^q \rightarrow \mathbb{R}^q$, and suppose that each component of the mapping $\mathbf{f}$ is differentiable with respect to its argument at the equilibrium point $\mathbf{x}_0 \in \mathbb{R}^q$, and let $\mathbf{y}_0 = \mathbf{f}(\mathbf{x}_0)$. Then there exist open sets $\mathcal{U} \subseteq \mathbb{R}^q$ containing $\mathbf{x}_0$ and $\mathcal{V} \subseteq \mathbb{R}^q$ containing $\mathbf{y}_0$ such that $\mathbf{f}$ is a diffeomorphism of $\mathcal{U}$ onto $\mathcal{V}$. If, in addition $\mathbf{f}$ is smooth, then the inverse mapping $\mathbf{f}^{-1}: \mathbb{R}^q \rightarrow \mathbb{R}^q$ is also smooth, that is, $\mathbf{f}$ is a smooth diffeomorphism.

The mapping $\mathbf{f}: \mathcal{U} \rightarrow \mathcal{V}$ is said to be a *diffeomorphism* of $\mathcal{U}$ onto $\mathcal{V}$ if it satisfies the following three conditions:

1. $\mathbf{f}(\mathcal{U}) = \mathcal{V}$.
2. The mapping $\mathbf{f}: \mathcal{U} \rightarrow \mathcal{V}$ is one-to-one (i.e., invertible).
3. Each component of the inverse mapping $\mathbf{f}^{-1}: \mathcal{V} \rightarrow \mathcal{U}$ is continuously differentiable with respect to its argument.

Returning to the issue of controllability, we may identify $\mathbf{f}(\mathcal{U}) = \mathcal{V}$ in the inverse function theorem with the mapping defined in Eq. (15.25). By using the inverse function theorem, we may say that if the controllability matrix $\mathbf{M}_c$ is of rank q , then locally there exists an inverse mapping defined by

$$(\mathbf{x}(n), \mathbf{x}(n+q)) = \mathbf{G}^{-1}(\mathbf{x}(n), \mathbf{u}_q(n)) \quad (15.27)$$

Equation (15.27), in effect, states that there exists an input sequence $\{\mathbf{u}_q(n)\}$ that can locally drive the network from state $\mathbf{x}(n)$ to state $\mathbf{x}(n+q)$ in q time steps. Accordingly, we may formally state the *local controllability theorem* as follows (Levin and Narendra, 1993):

Let a recurrent network be defined by Eqs. (15.16) and (15.17), and let its linearized version around the origin (i.e., equilibrium point) be defined by Eqs. (15.19) and (15.20). If the linearized system is controllable, then the recurrent network is locally controllable around the origin.

Local Observability

Using the linearized equations (15.19) and (15.20) repeatedly, we may write

$$\begin{aligned} \delta \mathbf{y}(n) &= \mathbf{c}^T \delta \mathbf{x}(n) \\ \delta \mathbf{y}(n+1) &= \mathbf{c}^T \delta \mathbf{x}(n+1) \\ &= \mathbf{c}^T \mathbf{A} \delta \mathbf{x}(n) + \mathbf{c}^T \mathbf{b} \delta u(n) \\ &\vdots \\ \delta \mathbf{y}(n+q-1) &= \mathbf{c}^T \mathbf{A}^{q-1} \delta \mathbf{x}(n) + \mathbf{c}^T \mathbf{A}^{q-2} \mathbf{b} \delta u(n) + \cdots + \mathbf{c}^T \mathbf{A} \mathbf{b} \delta u(n+q-3) \\ &\quad + \mathbf{c}^T \mathbf{b} \delta u(n+q-2) \end{aligned}$$

where q is the dimensionality of the state space. Accordingly, we may state that (Levin and Narendra, 1993):

The linearized system described by Eqs. (15.19) and (15.20) is observable if the matrix

$$\mathbf{M}_o = [\mathbf{c}, \mathbf{c} \mathbf{A}^T, \dots, \mathbf{c} (\mathbf{A}^T)^{q-1}] \quad (15.28)$$

is of rank q , that is, full rank.

The matrix $\mathbf{M}_o$ is called the *observability matrix* of the linearized system.

Let the recurrent network described by Eqs. (15.16) and (15.17) be driven by sequence of inputs defined by

$$\mathbf{u}_{q-1}(n) = [u(n), u(n+1), \dots, u(n+q-2)]^T \quad (15.29)$$

Correspondingly, let

$$\mathbf{y}_q(n) = [y(n), y(n+1), \dots, y(n+q-1)]^T \quad (15.30)$$

denote the vector of outputs produced by the initial state $\mathbf{x}(n)$ and the sequence of inputs $\mathbf{u}_{q-1}(n)$. We may then consider the mapping:

$$\mathbf{H}(\mathbf{u}_{q-1}(n), \mathbf{x}(n)) = (\mathbf{u}_{q-1}(n), \mathbf{y}_q(n)) \quad (15.31)$$

where $\mathbf{H}: \mathbb{R}^{2q-1} \rightarrow \mathbb{R}^{2q-1}$. In Problem 15.5 it is shown that the Jacobian of $\mathbf{y}_q(n)$ with respect to $\mathbf{x}(n)$, evaluated at the origin, is equal to the observability matrix $\mathbf{M}_o$ of Eq. (15.28). We may thus express the Jacobian of $\mathbf{H}$ with respect to $\mathbf{u}_{q-1}(n)$ and $\mathbf{x}(n)$ evaluated at the origin $(\mathbf{0}, \mathbf{0})$, as follows:

$$\begin{aligned} \mathbf{J}_{(\mathbf{0}, \mathbf{0})}^{(\omega)} &= \begin{bmatrix} \left(\frac{\partial \mathbf{u}_{q-1}(n)}{\partial \mathbf{u}_{q-1}(n)} \right)_{(\mathbf{0}, \mathbf{0})} & \left(\frac{\partial \mathbf{y}_q(n)}{\partial \mathbf{u}_{q-1}(n)} \right)_{(\mathbf{0}, \mathbf{0})} \\ \left(\frac{\partial \mathbf{u}_{q-1}(n)}{\partial \mathbf{x}(n)} \right)_{(\mathbf{0}, \mathbf{0})} & \left(\frac{\partial \mathbf{y}_q(n)}{\partial \mathbf{x}(n)} \right)_{(\mathbf{0}, \mathbf{0})} \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{I} & \mathbf{X} \\ \mathbf{0} & \mathbf{M}_o \end{bmatrix} \end{aligned} \quad (15.32)$$

where again the entry $\mathbf{X}$ is of no interest. The determinant of the Jacobian $\mathbf{J}_{(\mathbf{0}, \mathbf{0})}^{(\omega)}$ is equal to the product of the determinant of the identity matrix $\mathbf{I}$ (which equals 1) and the determinant of $\mathbf{M}_o$. If $\mathbf{M}_o$ is of full rank, then so is $\mathbf{J}_{(\mathbf{0}, \mathbf{0})}^{(\omega)}$. Invoking the inverse function theorem, we may therefore say that if the observability matrix $\mathbf{M}_o$ of the linearized system is of full rank, then locally there exists an inverse mapping defined by

$$(\mathbf{u}_{q-1}(n), \mathbf{x}(n)) = \mathbf{H}^{-1}(\mathbf{u}_{q-1}(n), \mathbf{y}_q(n)) \quad (15.33)$$

In effect, this equation states that in the local neighborhood of the origin, $\mathbf{x}(n)$ is some nonlinear function of both $\mathbf{u}_{q-1}(n)$ and $\mathbf{y}_q(n)$, and that nonlinear function is an observer of the recurrent network. We may therefore formally state the *local observability theorem* as follows (Levin and Narendra, 1993):

Let a recurrent network be defined by Eqs. (15.16) and (15.17), and let its linearized version around the origin (i.e., equilibrium point) be defined by Eqs. (15.19) and (15.20). If the linearized system is observable, then the recurrent network is locally observable around the origin.

Example 15.2

Consider a state-space model with matrix $\mathbf{A} = a\mathbf{I}$, where a is a scalar and $\mathbf{I}$ is the identity matrix. Then the controllability matrix $\mathbf{M}_c$ of Eq. (15.23) reduces to

$$\mathbf{M}_c = a[\mathbf{b}, \dots, \mathbf{b}, \mathbf{b}]$$

The rank of this matrix is 1. Hence, the linearized system with this value of matrix $\mathbf{A}$ is not controllable.

Putting $\mathbf{A} = a\mathbf{I}$ in Eq. (15.28), we obtain the observability matrix

$$\mathbf{M}_o = a[\mathbf{c}, \mathbf{c}, \dots, \mathbf{c}]$$

whose rank is also 1. The linearized system is also not observable.

NONLINEAR AUTOGRESSIVE WITH EXOGENOUS INPUTS MODEL

Consider a recurrent network with a single input and single output, whose behavior is described by the state equations (15.16) and (15.17). Given this state-space model, we wish to modify it into an input-output model as an equivalent representation of the recurrent network.

Using Eqs. (15.16) and (15.17), we may readily show that the output $y(n+q)$ is expressible in terms of the state $\mathbf{x}(n)$ and the vector of inputs $\mathbf{u}_q(n)$ as follows (see Problem 15.8):

$$y(n+q) = \Phi(\mathbf{x}(n), \mathbf{u}_q(n)) \quad (15.34)$$

where q is the dimensionality of the state space, and $\Phi: \mathbb{R}^{2q} \rightarrow \mathbb{R}$. Provided that the recurrent network is observable, we may use the local observability theorem to write

$$\mathbf{x}(n) = \Psi(\mathbf{y}_q(n), \mathbf{u}_{q-1}(n)) \quad (15.35)$$

where $\Psi: \mathbb{R}^{2q-1} \rightarrow \mathbb{R}^q$. Hence, substituting Eq. (15.35) in (15.34), we get

$$\begin{aligned} y(n+q) &= \Phi(\Psi(\mathbf{y}_q(n), \mathbf{u}_{q-1}(n)), \mathbf{u}_q(n)) \\ &= F(\mathbf{y}_q(n), \mathbf{u}_q(n)) \end{aligned} \quad (15.36)$$

where $\mathbf{u}_{q-1}(n)$ is contained in $\mathbf{u}_q(n)$ as its first $(q-1)$ elements, and the nonlinear mapping $F: \mathbb{R}^{2q} \rightarrow \mathbb{R}$ takes care of both Φ and Ψ . Using the definitions of $\mathbf{y}_q(n)$ and $\mathbf{u}_q(n)$ given in Eqs. (15.30) and (15.29), we may rewrite Eq. (15.36) in the expanded form:

$$y(n+q) = F(y(n+q-1), \dots, y(n), u(n+q-1), \dots, u(n))$$

Replacing n with $n-q+1$, we may equivalently write (Narendra, 1995):

$$y(n+1) = F(y(n), \dots, y(n-q+1), u(n), \dots, u(n-q+1)) \quad (15.37)$$

Stated in words, some nonlinear mapping $F: \mathbb{R}^{2q} \rightarrow \mathbb{R}$ exists whereby the present value of the output $y(n+1)$ is uniquely defined in terms of its past values $y(n), \dots, y(n-q+1)$ and the present and past values of the input $u(n), \dots, u(n-q+1)$. For this input-output representation to be equivalent to the state-space model of Eqs. (15.16)

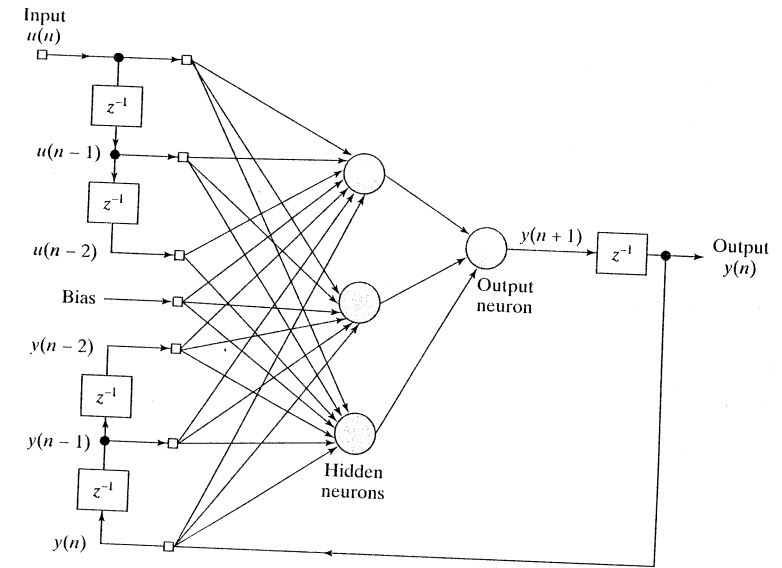


FIGURE 15.7 NARX network with $q = 3$ hidden neurons.

and (15.17), the recurrent network must be observable. The practical implication of this equivalence is that the NARX model of Fig. 15.1, with its global feedback limited to the output neuron, is in fact able to simulate the corresponding fully recurrent state-space model of Fig. 15.2 (assuming that $m = 1$ and $p = 1$) with no difference between their input-output behavior.

Example 15.3

Consider again the fully connected recurrent network of Fig. 15.6. For the purpose of our present discussion, suppose that one of the inputs, $u_2(n)$ say, is reduced to zero, so that we have a single input, single output network. We may then replace this fully connected recurrent network by the NARX model shown in Fig. 15.7, provided that the network is locally observable. This equivalence holds despite the fact that the NARX model has limited feedback that originates only from the output neuron, whereas in the fully connected recurrent network of Fig. 15.6 the feedback around the multilayer perceptron originates from the three hidden/output neurons.

15.5 COMPUTATIONAL POWER OF RECURRENT NETWORKS

Recurrent networks, exemplified by the state-space model of Fig. 15.2 and the NARX model of Fig. 15.1, have an inherent ability to simulate finite-state automata. Automata represent abstractions of information processing devices such as computers. Indeed,

automata and neural networks share a long history.⁶ In his 1967 book (p. 55), Minsky makes the following consequential statement:

“Every finite-state machine is equivalent to, and can be ‘simulated’ by, some neural net. That is, given any finite-state machine $\mathcal{M}$, we can build a certain neural net $\mathcal{N}^{\mathcal{M}}$ which, regarded as a black-box machine, will behave precisely like $\mathcal{M}$!”

The early work on recurrent networks used hard threshold logic for the activation function of a neuron rather than soft sigmoid function.

Perhaps the first experimental demonstration of whether or not a recurrent network could learn the contingencies implied by a small finite-state grammar was reported in Cleeremans et al. (1989). Specifically, the simple recurrent network (Fig. 15.3) was presented with strings derived from the grammar and required to predict the next letter at every step. The predictions were context dependent since each letter appeared twice in the grammar and was followed in each case by different successors. It was shown that the network is able to develop internal representations in its hidden neurons that correspond to the states of the automaton (finite-state machine). In Kremer (1995) a formal proof is presented that the simple recurrent network has a computational power as great as that of any finite-state machine.

In a generic sense, the computational power of a recurrent network is embodied in two main theorems:

Theorem I (Siegelmann and Sontag, 1991).

All Turing machines may be simulated by fully connected recurrent networks built on neurons with sigmoid activation functions.

The *Turing machine* is an abstract computing device invented by Turing (1936). It consists of three functional blocks as depicted in Fig. 15.8: (1) *control unit* that can assume any one of a finite number of possible states; (2) *linear tape* (assumed to be infinite in both directions) that is marked off into discrete squares with each square available to store a single symbol taken from a finite set of symbols; and (3) *read-write head* that moves along the tape and transmits information to and from the control unit (Fischler and Firschein, 1987). For the present discussion it suffices to say that the Turing machine is an abstraction that is functionally as powerful as any computer. This idea is known as the *Church–Turing hypothesis*.

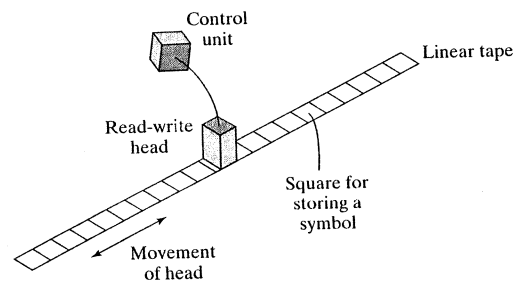


FIGURE 15.8 Turing Machine.

Theorem II (Siegelmann et al., 1997)

NARX networks with one layer of hidden neurons with bounded, one-sided saturate activation functions and a linear output neuron can simulate fully connected recurrent networks with bounded, one-sided saturated activation functions, except for a linear slow down.

A “linear slowdown” means that if the fully connected recurrent network with N neurons computes a task of interest in time T , then the total time taken by the equivalent NARX network is $(N + 1)T$. A function $\varphi(\cdot)$ is said to be a *bounded, one-sided saturated (BOSS) function* if it satisfies the following three conditions:

1. The function $\varphi(\cdot)$ has a bounded range; that is, $a \leq \varphi(x) \leq b$, $a \neq b$, for all $x \in \mathbb{R}$.
2. The function $\varphi(\cdot)$ is saturated on the left side; that is, there exist values s and S such that $\varphi(x) = S$ for all $x \leq s$.
3. The function $\varphi(\cdot)$ is nonconstant; that is, $\varphi(x_1) \neq \varphi(x_2)$ for some x_1 and x_2 .

The threshold (Heaviside) and piecewise-linear functions satisfy the BOSS conditions. However, in a strict sense, a sigmoid function is not a BOSS function because it does not satisfy condition 2. Nevertheless, with a minor modification, it can be made into a BOSS function by writing (in the case of a logistic function)

$$\varphi(x) = \begin{cases} \frac{1}{1 + \exp(-x)} & \text{for } x > s \\ 0 & \text{for } x \leq s \end{cases}$$

where $s \in \mathbb{R}$. In effect, the logistic function is truncated for $x \leq s$.

As a corollary to Theorems I and II, we may state the following (Giles, 1996):

NARX networks with one hidden layer of neurons with BOSS activations functions and a linear output neuron are Turing equivalent.

Figure 15.9 presents a portrayal of Theorems I and II and this corollary. It should, however, be noted that when the network architecture is constrained, the computational power of a recurrent network may no longer hold, as described in Sperduti(1997). References to examples of constrained network architectures are presented in note 7.

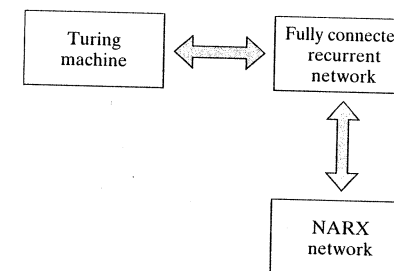


FIGURE 15.9 Illustration of Theorems I and II, and corollary to them.

EARNING ALGORITHMS

We now turn to the issue of training recurrent networks. From Chapter 4 we recall that there are two modes of training an ordinary (static) multilayer perceptron: batch mode and sequential mode. In the batch mode, the sensitivity of the network is computed for the entire training set before adjusting the free parameters of the network. In the sequential mode, on the other hand, parameter adjustments are made after the presentation of each pattern in the training set. Likewise, we have two modes of training a recurrent network, as described here (Williams and Zipser, 1995):

1. *Epochwise training.* For a given epoch, the recurrent network starts running from some initial state until it reaches a new state, at which point the training is stopped and the network is reset to an initial state for the next epoch. The initial state doesn't have to be the same for each epoch of training. Rather, what is important is for the initial state for the new epoch to be different from the state reached by the network at the end of the previous epoch. Consider, for example, the use of a recurrent network to emulate the operation of a finite-state machine, that is, a device whose distinguishable internal configurations (states) are finite in number. In such a situation it is reasonable to use epochwise training since we have a good possibility for a number of distinct initial states and a set of distinct final states in the machine to be emulated by the recurrent network. In epochwise training for recurrent networks the term "epoch" is used in a sense different from that for an ordinary multilayer perceptron. In the current terminology, the epoch for the recurrent network corresponds to one training pattern for the ordinary multilayer perceptron.

2. *Continuous training.* This second method of training is suitable for situations where there are no reset states available and/or on-line learning is required. The distinguishing feature of continuous training is that the network learns while signal processing is being performed by the network. Simply put, the learning process never stops. Consider, for example, the use of a recurrent network to model a nonstationary process such as a speech signal. In this kind of situation, continuous operation of the network offers no convenient times at which to stop the training and begin anew with different values for the free parameters of the network.

Keeping these two modes of training in mind, in the next two sections we will describe different learning algorithms for recurrent networks as summarized here:

- The back-propagation-through-time algorithm, discussed in Section 15.7, operates on the premise that the temporal operation of a recurrent network may be unfolded into a multilayer perceptron. This would then pave the way for application of the standard back-propagation algorithm. Back-propagation through time can be implemented in the epochwise mode, continuous (real-time) mode, or combination thereof.
- The real-time recurrent learning algorithm, discussed in Section 15.8, is derived from the state-space model described by Eqs. (15.10) and (15.11).

These two algorithms share many common features. First, they are both based on the method of gradient descent, whereby the instantaneous value of a cost function (based

on a squared-error criterion) is minimized with respect to the synaptic weights of the network. Second, they are both relatively simple to implement, but can be slow to converge. Third, they are related in that the signal-flow graph representation of the back-propagation-through-time algorithm can be obtained from *transposition* of the signal-flow graph representation of a certain form of the real-time recurrent learning algorithm (Lefebvre, 1991; Beaufays and Wan, 1994).

Real-time (continuous) learning, based on gradient descent, uses the *minimum amount of available information*, namely an instantaneous estimate of the gradient of the cost function with respect to the parameter vector to be adjusted. We may accelerate the learning process by exploiting Kalman filter theory, which utilizes information contained in the training data more effectively. In Section 15.10 we describe the decoupled extended Kalman filter, by means of which we are able to tackle dynamic learning tasks that would be very difficult for gradient-descent based methods. A brief review of Kalman filters is presented in Section 15.9. Note that the decoupled extended Kalman filter is applicable to both static feedforward networks as well as recurrent networks.

Some Heuristics

Before proceeding to describe the new learning algorithms mentioned, we will list some heuristics for the improved training of recurrent networks that involve the use of gradient-descent methods (Giles, 1996):

- Lexigraphic order of training samples should be followed, with the shortest strings of symbols being presented to the network first.
- The training should begin with a small training sample, and then its size should be incrementally increased as the training proceeds.
- The synaptic weights of the network should be updated only if the absolute error on the training sample currently being processed by the network is greater than some prescribed criterion.
- The use of weight decay during training is recommended; weight decay, a crude form of complexity regularization, is discussed in Chapter 4.

The first heuristic is of particular interest. If implementable, it provides a procedure for alleviating the vanishing gradients problem that arises in recurrent networks trained by means of gradient-descent methods. This problem is discussed in Section 15.12.

15.7 BACK-PROPAGATION THROUGH TIME

The *back-propagation-through-time (BPTT) algorithm* for training a recurrent network is an extension of the standard back-propagation algorithm.⁸ It may be derived by *unfolding* the temporal operation of the network into a layered feedforward network, the topology of which grows by one layer at every time step.

To be specific, let N denote a recurrent network required to learn a temporal task, starting from time n_0 all the way up to time n . Let N^* denote the feedforward network

that results from unfolding the temporal operation of the recurrent network $\mathcal{N}$. The unfolded network $\mathcal{N}^*$ is related to the original network $\mathcal{N}$ as follows:

1. For each time step in the interval $(n_0, n]$, the network $\mathcal{N}^*$ has a layer containing K neurons, where K is the number of neurons contained in the network $\mathcal{N}$.
2. In every layer of the network $\mathcal{N}^*$ there is a copy of each neuron in the network $\mathcal{N}$.
3. For each time step $l \in [n_0, n]$, the synaptic connection from neuron i in layer l to neuron j in layer $l + 1$ of the network $\mathcal{N}^*$ is a copy of the synaptic connection from neuron i to neuron j in the network $\mathcal{N}$.

These points are illustrated in the following example.

Example 15.4

Consider the two-neuron recurrent network $\mathcal{N}$ shown in Fig. 15.10a. To simplify the presentation we have omitted unit delay operators z^{-1} that should be inserted in each of the synaptic connections (including the self-loops) in Fig. 15.10a. By unfolding the temporal operation of this network in a step-by-step manner, we get the signal-flow graph shown in Fig. 15.10b where the starting time $n_0 = 0$. The graph of Fig. 15.10b represents the layered feedforward $\mathcal{N}^*$, where a new layer is added at each step of the temporal operation. ■

Application of the unfolding procedure leads to two basically different implementations of back-propagation through time, depending on whether epochwise training or continuous (real-time) training is used. These two methods of recurrent learning are now described in that order.

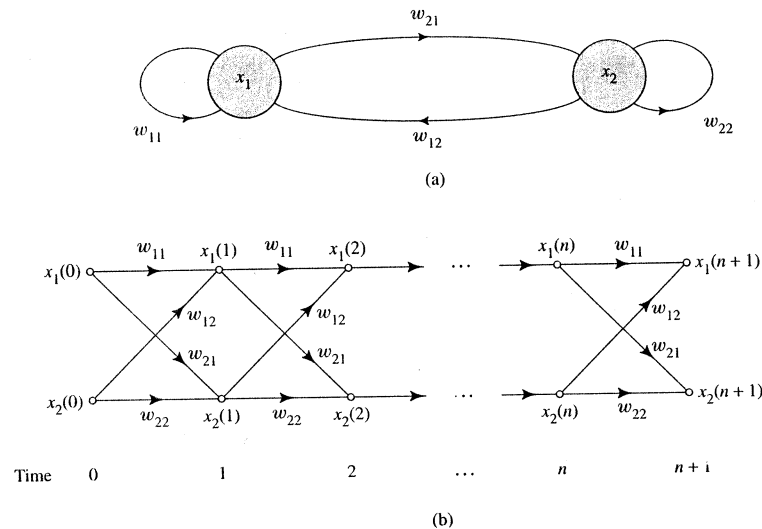


FIGURE 15.10 (a) Architectural graph of a two-neuron recurrent network $\mathcal{N}$. (b) Signal-flow graph of the network $\mathcal{N}$ unfolded in time.

Epochwise Back-propagation through Time

Let the data set used to train a recurrent network be partitioned into independent epochs, with each epoch representing a temporal pattern of interest. Let n_0 denote the start time of an epoch and n_1 denote its end time. Given this epoch, we may define the cost function

$$\mathcal{E}_{\text{total}}(n_0, n_1) = \frac{1}{2} \sum_{n=n_0}^{n_1} \sum_{j \in \mathcal{A}} e_j^2(n) \quad (15.38)$$

where $\mathcal{A}$ is the set of indices j pertaining to those neurons in the network for which desired responses are specified, and $e_j(n)$ is the error signal at the output of such a neuron measured with respect to some desired response. We wish to compute sensitivity of the network, that is, the partial derivatives of the cost function $\mathcal{E}_{\text{total}}(n_0, n_1)$ with respect to synaptic weights of the network. To do so, we may use the *epochwise back-propagation-through-time (BPTT) algorithm*, which builds on the batch mode of standard back-propagation learning that is described in Chapter 4. The epochwise BPTT algorithm proceeds as follows (Williams and Peng, 1990):

- First, a single forward pass of the data through the network for the interval $(n_0, n_1]$ is performed. The complete record of input data, network state (i.e., synaptic weights of the network), and desired responses over this interval is *saved*.
- A single backward pass over this past record is performed to compute the values of the local gradients

$$\delta_j(n) = -\frac{\partial \mathcal{E}_{\text{total}}(n_0, n_1)}{\partial v_j(n)} \quad (15.39)$$

for all $j \in \mathcal{A}$ and $n_0 < n \leq n_1$. This computation is performed by using the formula:

$$\delta_j(n) = \begin{cases} \varphi'(v_j(n))e_j(n) & \text{for } n = n_1 \\ \varphi'(v_j(n)) \left[e_j(n) + \sum_{k \in \mathcal{A}} w_{jk} \delta_k(n+1) \right] & \text{for } n_0 < n < n_1 \end{cases} \quad (15.40)$$

where $\varphi'(\cdot)$ is the derivative of an activation function with respect to its argument, and $v_j(n)$ is the induced local field of neuron j . It is assumed that all neurons in the network have the same activation function $\varphi(\cdot)$. The use of Eq. (15.40) is repeated, starting from time n_1 and working back, step by step, to time n_0 ; the number of steps involved here is equal to the number of time steps contained in the epoch.

- Once the computation of back-propagation has been performed back to time $n_0 + 1$, the following adjustment is applied to the synaptic weight w_{ji} of neuron j :

$$\begin{aligned} \Delta w_{ji} &= -\eta \frac{\partial \mathcal{E}_{\text{total}}(n_0, n_1)}{\partial w_{ji}} \\ &= \eta \sum_{n=n_0+1}^{n_1} \delta_j(n) x_i(n-1) \end{aligned} \quad (15.41)$$

where η is the learning-rate parameter and $x_i(n-1)$ is the input applied to the i th synapse of neuron j at time $n-1$.

Comparing the procedure just described for epochwise BPTT with the batch mode of standard back-propagation learning, we see that the basic difference between them is that in the former case the desired responses are specified for neurons in many layers of the network because the actual output layer is replicated many times when the temporal behavior of the network is unfolded.

Truncated Back-propagation through Time

To use back-propagation through time in a real-time fashion, we use the instantaneous value of the sum of squared errors, namely,

$$\mathcal{E}(n) = \frac{1}{2} \sum_{j \in \mathcal{A}} e_j^2(n)$$

as the cost function to be minimized. As with the sequential (stochastic) mode of standard back-propagation learning, we use the negative gradient of the cost function $\mathcal{E}(n)$ to compute the appropriate adjustments to the synaptic weights of the network at each time instant n . The adjustments are made on a continuous basis, while the network is running. However, in order to do this in a computationally feasible manner, we only save the relevant history of input data and network state for a fixed number of time steps, called the *truncation depth*. Henceforth the truncation depth is denoted by h . Any information older than h time steps into the past is considered irrelevant, and may therefore be ignored. If we were not to truncate the computation, thereby permitting it to go back to the starting time, the computation time and storage requirement would grow linearly with time as the network runs, eventually reaching a point where the whole learning process becomes impractical.

This second form of the algorithm is called the *truncated back-propagation-through-time (BPTT(h)) algorithm* (Williams and Peng, 1990). The local gradient for neuron j is now defined by

$$\delta_j(l) = -\frac{\partial \mathcal{E}(l)}{\partial v_j(l)} \quad \begin{array}{ll} \text{for all } j \in \mathcal{A} & \\ \text{and } n-h < l \leq n & \end{array} \quad (15.42)$$

which in turn leads to the formula:

$$\delta_j(l) = \begin{cases} \varphi'(v_j(l))e_j(l) & \text{for } l = n \\ \varphi'(v_j(l)) \sum_{k \in \mathcal{A}} w_{kj}(l) \delta_k(l+1) & \text{for } n-h < l < n \end{cases} \quad (15.43)$$

Once the computation of back-propagation has been performed back to time $n-h+1$, the following adjustment is applied to the synaptic weight w_{ji} of neuron j :

$$\Delta w_{ji}(n) = \eta \sum_{l=n-h+1}^n \delta_j(l)x_i(l-1) \quad (15.44)$$

where η and $x_i(l-1)$ are as defined previously. Note that the use of $w_{kj}(l)$ in Eq. (15.43) requires that a history of weight values be maintained. The use of w_{kj} in this equation may be justified only if the learning-rate parameter η is small enough to ensure that the weight values do not change significantly from one time step to the next.

In comparing Eq. (15.43) with (15.40), we see that, unlike the epochwise BPTT algorithm, the error signal is only injected into the computation at the current time n . This explains the reason for not keeping a record of past values of the desired responses. In effect, the truncated back-propagation-through-time algorithm treats the computation for all earlier time steps similar to the way in which the stochastic back-propagation algorithm (discussed in Chapter 4) treats the computations for hidden neurons in a multilayer perceptron.

Some Practical Considerations

In real-life applications of BPTT, the use of truncation is not as artificial as it may sound. Unless the recurrent network is unstable, there should be a convergence of the derivatives $\partial \mathcal{E}(l)/\partial v_j(l)$ because computations farther back in time correspond to higher powers of feedback strengths (roughly equal to sigmoid slopes multiplied by weights). In any event, the truncation depth h must be large enough to produce derivatives that closely approximate the actual values. This requirement places a lower bound on the value of h . For example, in the application of dynamically driven recurrent networks to engine idle-speed control, the value $h=30$ is considered to be a reasonably conservative choice for that learning task to be accomplished (Puskorius et al., 1996).

One other practical matter needs to be discussed. The unfolding procedure described in this section for back-propagation through time provides a useful tool for picturing it in terms of a cascade of similar layers progressing forward in time, thereby helping us to develop an understanding of how the procedure functions. This strong point is unfortunately the cause of its weakness. The procedure works perfectly fine for relatively simple recurrent networks consisting of a few neurons. However, the underlying formulas, particularly Eq. (15.43), become unwieldy when the unfolding procedure is applied to more general architectures that are typical of those encountered in practice. In situations of this kind, the preferred procedure is to use the more general approach described in Werbos (1990), in which each expression in the forward propagation of a layer gives rise to a corresponding set of back-propagation expressions. An advantage of this approach is its homogeneous treatment of forward and recurrent (feedback) connections.

To describe the mechanics of this particular form of BPTT(h), let F_{-x}^l denote an *ordered derivative* of the network output at node l with respect to x . To derive the back-propagation equations, the forward propagation equations are considered in reverse order. From each equation we derive one or more back-propagation expressions according to the following principle:

$$\text{If } a = \varphi(b, c), \text{ then } F_{-b}^l = \frac{\partial \varphi}{\partial b} F_{-a}^l \quad \text{and} \quad F_{-c}^l = \frac{\partial \varphi}{\partial c} F_{-a}^l \quad (15.45)$$

Example 15.5

To clarify the notion of ordered derivatives, consider a nonlinear system described by the following pair of equations:

$$x_1 = \log u + x_2^3$$

$$y = x_1^2 + 3x_2$$

The variable x_2 influences the output y in two ways: directly via the second equation, and indirectly via the first equation. The ordered derivative of y with respect to x_2 is defined by the total causal impact that includes the direct and indirect effects of x_2 on y , as shown by

$$\begin{aligned} F_{-x_2} &= \frac{\partial y}{\partial x_2} + \frac{\partial y}{\partial x_1} \frac{\partial x_1}{\partial x_2} \\ &= 3 + (2x_1)(3x_2^2) \\ &= 3 + 6x_1x_2^2 \end{aligned}$$

In programming the ordered derivatives for BPTT(h), the quantity on the right-hand side of each ordered derivative in Eq. (15.45) is added to the previous value of the left-hand side. In this way, the appropriate derivatives are distributed from a given node in the network to all the nodes and synaptic weights that feed it in the forward direction, with due allowance being made for any delays that may be present in each connection. The simplicity of the formulation described herein reduces the need for visualizations such as unfolding in time or signal-flow graphs. In Feldkamp and Puskorius (1998) and Puskorius et al. (1996), this procedure is used to develop a pseudocode for implementing the BPTT(h) algorithm.

3 REAL-TIME RECURRENT LEARNING

In this section we describe another learning algorithm referred to as *real-time recurrent learning (RTRL)*.⁹ The algorithm derives its name from the fact that adjustments are made to the synaptic weights of a fully connected recurrent network in real time, that is, while the network continues to perform its signal processing function (Williams and Zipser, 1989). Figure 15.11 shows the layout of such a recurrent network. It consists of q neurons with m external inputs. The network has two distinct layers: a *concatenated input-feedback layer* and a *processing layer of computation nodes*. Correspondingly, the synaptic connections of the network are made up of feedforward and feedback connections.

The state-space description of the network is defined by Eqs. (15.10) and (15.11). The process equation (15.10) is reproduced here in the following expanded form:

$$\mathbf{x}(n+1) = \begin{bmatrix} \varphi(\mathbf{w}_1^T \xi(n)) \\ \vdots \\ \varphi(\mathbf{w}_j^T \xi(n)) \\ \vdots \\ \varphi(\mathbf{w}_q^T \xi(n)) \end{bmatrix} \quad (15.46)$$

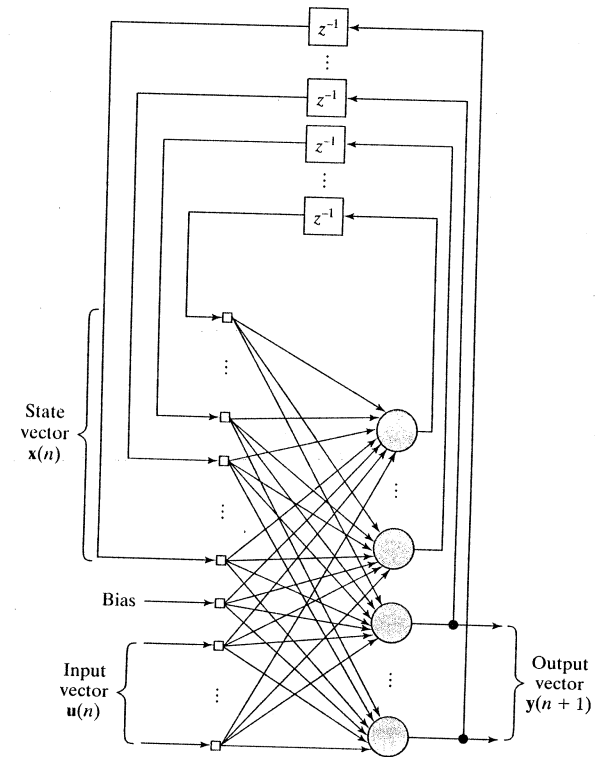


FIGURE 15.11 Fully connected recurrent network for formulation of the RTRL algorithm.

where it is assumed that all the neurons have a common activation function $\varphi(\cdot)$. The $(q+m+1)$ -by-1 vector $\mathbf{w}_j$ is the synaptic weight vector of neuron j in the recurrent network, that is,

$$\mathbf{w}_j = \begin{bmatrix} \mathbf{w}_{a,j} \\ \mathbf{w}_{b,j} \end{bmatrix}, \quad j = 1, 2, \dots, q \quad (15.47)$$

where $\mathbf{w}_{a,j}$ and $\mathbf{w}_{b,j}$ are the j th columns of the transposed weight matrices $\mathbf{W}_a^T$ and $\mathbf{W}_b^T$, respectively. The $(q+m+1)$ -by-1 vector $\xi(n)$ is defined by

$$\xi(n) = \begin{bmatrix} \mathbf{x}(n) \\ \mathbf{u}(n) \end{bmatrix} \quad (15.48)$$

where $\mathbf{x}(n)$ is the q -by-1 state vector and $\mathbf{u}(n)$ is the $(m+1)$ -by-1 input vector. The first element of $\mathbf{u}(n)$ is +1 and, in a corresponding way, the first element of $\mathbf{w}_{b,j}$ is equal to the bias b_j applied to neuron j .

To simplify the presentation, we introduce three new matrices $\Lambda_j(n)$, $U_j(n)$, and $\Phi(n)$, described as follows:

1. $\Lambda_j(n)$ is a q -by- $(q + m + 1)$ matrix defined as the partial derivative of the state vector $\mathbf{x}(n)$ with respect to the weight vector $\mathbf{w}_j$:

$$\Lambda_j(n) = \frac{\partial \mathbf{x}(n)}{\partial \mathbf{w}_j}, \quad j = 1, 2, \dots, q \quad (15.49)$$

2. $U_j(n)$ is a q -by- $(q + m + 1)$ matrix whose rows are all zero, except for the j th row that is equal to the transpose of vector $\xi(n)$:

$$U_j(n) = \begin{bmatrix} 0 \\ \xi^T(n) \\ 0 \end{bmatrix} \leftarrow j\text{th row}, \quad j = 1, 2, \dots, q \quad (15.50)$$

3. $\Phi(n)$ is a q -by- q diagonal matrix whose k th diagonal element is the partial derivative of the activation function with respect to its argument, evaluated at $\mathbf{w}_k^T \xi(n)$:

$$\Phi(n) = \text{diag}(\varphi'(\mathbf{w}_1^T \xi(n)), \dots, \varphi'(\mathbf{w}_q^T \xi(n))) \quad (15.51)$$

With these definitions, we may now differentiate Eq. (15.46) with respect to $\mathbf{w}_j$. Then, using the chain rule of calculus, we obtain the following recursive equation:

$$\Lambda_j(n+1) = \Phi(n)[W_a(n)\Lambda_j(n) + U_j(n)], \quad j = 1, 2, \dots, q \quad (15.52)$$

This recursive equation describes the *nonlinear state dynamics* (i.e., evolution of the state) of the real-time recurrent learning process.

To complete the description of this learning process, we need to relate the matrix $\Lambda_j(n)$ to the gradient of the error surface with respect to $\mathbf{w}_j$. To do this, we first use the measurement equation (15.11) to define the p -by-1 error vector:

$$\begin{aligned} \mathbf{e}(n) &= \mathbf{d}(n) - \mathbf{y}(n) \\ &= \mathbf{d}(n) - \mathbf{C}\mathbf{x}(n) \end{aligned} \quad (15.53)$$

The instantaneous sum of squared errors at time n is defined in terms of $\mathbf{e}(n)$ by

$$\mathcal{E}(n) = \frac{1}{2} \mathbf{e}^T(n) \mathbf{e}(n) \quad (15.54)$$

The objective of the learning process is to minimize a cost function obtained by summing $\mathcal{E}(n)$ over all time n ; that is,

$$\mathcal{E}_{\text{total}} = \sum_n \mathcal{E}(n)$$

To accomplish this objective we may use the method of steepest descent, which requires knowledge of the *gradient matrix*, written as

$$\begin{aligned} \nabla_{\mathbf{w}} \mathcal{E}_{\text{total}} &= \frac{\partial \mathcal{E}_{\text{total}}}{\partial \mathbf{W}} \\ &= \sum_n \frac{\partial \mathcal{E}(n)}{\partial \mathbf{W}} \\ &= \sum_n \nabla_{\mathbf{w}} \mathcal{E}(n) \end{aligned}$$

where $\nabla_{\mathbf{w}} \mathcal{E}(n)$ is the gradient of $\mathcal{E}(n)$ with respect to the weight matrix $\mathbf{W} = \{\mathbf{w}_k\}$. We may, if desired, continue with this equation and derive update equations for the synaptic weights of the recurrent network without invoking approximations. However, in order to develop a learning algorithm that can be used to train the recurrent network in *real time*, we must use an instantaneous *estimate* of the gradient, namely $\nabla_{\mathbf{w}} \mathcal{E}(n)$, which results in an *approximation* to the method of steepest descent.

Returning to Eq. (15.54) as the cost function to be minimized, we differentiate it with respect to the weight vector $\mathbf{w}_j$, obtaining

$$\begin{aligned} \frac{\partial \mathcal{E}(n)}{\partial \mathbf{w}_j} &= \left(\frac{\partial \mathbf{e}(n)}{\partial \mathbf{w}_j} \right) \mathbf{e}(n) \\ &= -\mathbf{C} \left(\frac{\partial \mathbf{x}(n)}{\partial \mathbf{w}_j} \right) \mathbf{e}(n) \\ &= -\mathbf{C} \Lambda_j(n) \mathbf{e}(n), \quad j = 1, 2, \dots, q \end{aligned} \quad (15.55)$$

The adjustment applied to synaptic weight vector $\mathbf{w}_j(n)$ of neuron j is therefore determined by

$$\begin{aligned} \Delta \mathbf{w}_j(n) &= -\eta \frac{\partial \mathcal{E}(n)}{\partial \mathbf{w}_j} \\ &= \eta \mathbf{C} \Lambda_j(n) \mathbf{e}(n), \quad j = 1, 2, \dots, q \end{aligned} \quad (15.56)$$

where η is the learning-rate parameter and $\Lambda_j(n)$ is itself governed by Eq. (15.52).

The only remaining item is that of specifying the *initial conditions* to start the learning process. For this purpose we set

$$\Lambda_j(0) = \mathbf{0} \quad \text{for all } j \quad (15.57)$$

the implication of which is that initially the recurrent network resides in a constant state.

Table 15.1 presents a summary of the real-time recurrent learning algorithm. The formulation of the algorithm as described here applies to an arbitrary activation function $\varphi(\cdot)$ that is differentiable with respect to its argument. For the special case of a sigmoidal nonlinearity in the form of a hyperbolic tangent function, we have

$$\begin{aligned} x_j(n+1) &= \varphi(v_j(n)) \\ &= \tanh(v_j(n)) \end{aligned}$$

TABLE 15.1 Summary of the Real-Time Recurrent Learning Algorithm

Parameters:

- m = dimensionality of input space
- q = dimensionality of state space
- p = dimensionality of output space
- $\mathbf{w}_j$ = synaptic weight vector of neuron $j, j = 1, 2, \dots, q$.

Initialization:

1. Set the synaptic weights of the algorithm to small values selected from a uniform distribution.
2. Set the initial value of the state vector $\mathbf{x}(0) = \mathbf{0}$.
3. Set $\Lambda_j(0) = \mathbf{0}$ for $j = 1, 2, \dots, q$.

Computations: Compute for $n = 0, 1, 2, \dots$,

$$\Lambda_j(n+1) = \Phi(n)[\mathbf{W}_a(n)\Lambda_j(n) + \mathbf{U}_j(n)]$$

$$\mathbf{e}(n) = \mathbf{d}(n) - \mathbf{C}\mathbf{x}(n)$$

$$\Delta \mathbf{w}_j(n) = \eta \mathbf{C} \Lambda_j(n) \mathbf{e}(n)$$

The definitions of $\mathbf{x}(n)$, $\Lambda_j(n)$, $\mathbf{U}_j(n)$, and $\Phi(n)$ are given in Eqs. (15.46), (15.49), (15.50), and (15.51), respectively.

and

$$\begin{aligned} \varphi'(v_j(n)) &= \frac{\partial \varphi(v_j(n))}{\partial v_j(n)} \\ &= \text{sech}^2(v_j(n)) \\ &= 1 - x_j^2(n+1) \end{aligned} \quad (15.58)$$

where $v_j(n)$ is the induced local field of neuron j and $x_j(n+1)$ is its state at $n+1$.

The use of the instantaneous gradient $\nabla_{\mathbf{w}} \mathcal{E}(n)$ means that the real-time recurrent learning algorithm described here deviates from a non-real-time one based on the true gradient $\nabla_{\mathbf{w}} \mathcal{E}_{\text{total}}$. However, this deviation is exactly analogous to that encountered in the standard back-propagation algorithm used in Chapter 4 to train an ordinary multilayer perceptron, where weight changes are made after each pattern presentation. While the real-time recurrent learning algorithm is not guaranteed to follow the precise negative gradient of the total error function $\mathcal{E}_{\text{total}}(\mathbf{W})$ with respect to the weight matrix $\mathbf{W}$, the practical differences between the real-time and non-real-time versions are often slight; these two versions become nearly identical as the learning-rate parameter η is reduced. The most severe potential consequence of this deviation from the true gradient-following behavior is that the observed trajectory (obtained by plotting $\mathcal{E}(n)$ versus the elements of the weight matrix $\mathbf{W}(n)$) may itself depend on the weight changes produced by the algorithm, which may be viewed as another source of feedback and therefore a cause of instability in the system. We can avoid this effect by using a learning-rate parameter η small enough to make the time

scale of the weight changes much smaller than the time scale of the network operation (Williams and Zipser, 1989).

Example 15.6

In this example we formulate the RTRL algorithm for the fully recurrent network shown in Fig. 15.6 with two inputs and single output. The network has three neurons, with the composition of matrices $\mathbf{W}_a$, $\mathbf{W}_b$, and $\mathbf{C}$ as described in Example 15.1.

With $m = 2$ and $q = 3$, we find from Eq. (15.48) that

$$\xi(n) = \begin{bmatrix} x_1(n) \\ x_2(n) \\ x_3(n) \\ 1 \\ u_1(n) \\ u_2(n) \end{bmatrix}$$

Let $\lambda_{j,kl}(n)$ denote the kl -th element of matrix $\Lambda_j(n)$. The use of Eqs. (15.52) and (15.56) then yields, respectively,

$$\lambda_{j,kl}(n+1) = \varphi'(v_j(n)) \left[\sum_{i=1}^3 w_{ji}(n) \lambda_{i,kl}(n) + \delta_{kj} \xi_l(n) \right]$$

$$\Delta w_{ki}(n) = \eta (d_i(n) - x_i(n)) \lambda_{1,kl}(n)$$

where δ_{kj} is the Kronecker delta, which is equal to 1 for $k = j$ and zero otherwise; and $(j, k) = 1, 2, 3$ and $l = 1, 2, \dots, 6$. Figure 15.12 presents a *sensitivity graph* determining the evolution of the weight adjustment $\Delta w_{ki}(n)$. Note that $\mathbf{W}_a = \{w_{ji}\}$ for $(j, i) = 1, 2, 3$, and $\mathbf{W}_b = \{w_{ji}\}$ for $j = 1, 2, 3$ and $l = 4, 5, 6$.

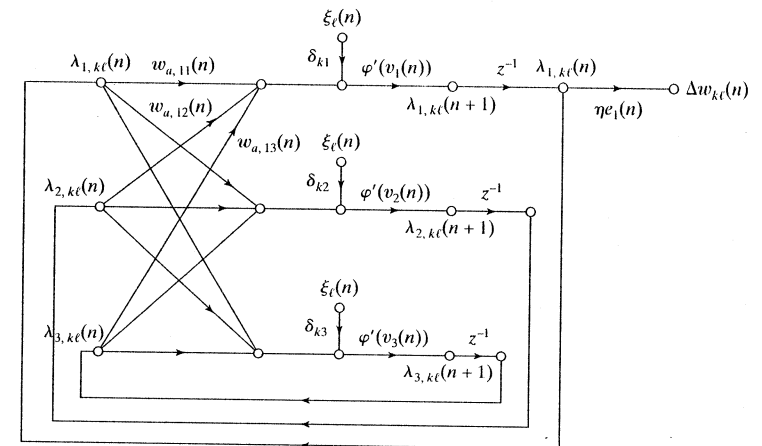


FIGURE 15.12 Sensitivity graph of the fully recurrent network of Fig. 15.6. Note: The three nodes labeled $\xi_i(n)$ are all to be viewed as a single input.

Teacher Forcing

A strategy that is frequently used in the training of recurrent networks is *teacher forcing* (Williams and Zipser, 1989, 1995); in adaptive filtering, teacher forcing is known as the *equation-error method* (Mendel, 1995). Basically, teacher forcing involves replacing the actual output of a neuron, during training of the network, with the corresponding desired response (i.e., target signal) in subsequent computation of the dynamic behavior of the network, whenever that desired response is available. Although teacher forcing is being described under the RTRL algorithm, its use applies to any other learning algorithm. For it to be applicable, however, the neuron in question must feed its output back to the network.

Beneficial effects of teacher forcing include (Williams and Zipser, 1995):

- *Teacher forcing may lead to faster training.* The reason for this improvement is the use of teacher forcing amounts to the assumption that the network has correctly learned all the earlier parts of the task that pertain to the neurons where teacher forcing has been applied.
- *Teacher forcing may serve as a corrective mechanism during training.* For example, the synaptic weights of the network may have the correct values, but somehow the network is currently operating in the wrong region of the state space. Clearly, adjusting the synaptic weights is the wrong strategy in such a situation.

A gradient-based learning algorithm that uses teacher forcing is in actual fact optimizing a cost function different from its unforced counterpart. The teacher forced and unforced versions of the algorithm may therefore yield different solutions, unless the pertinent error signals are zero, in which case learning is unnecessary.

1 KALMAN FILTERS

As mentioned previously, continuous learning based on gradient descent, exemplified by the real-time recurrent learning algorithm, is typically slow due to reliance on instantaneous estimates of gradients. We may overcome this serious limitation by viewing the supervised training of a recurrent network as an *optimum filtering problem*, the solution of which *recursively* utilizes information contained in the training data in a manner going back to the first iteration of the learning process. The idea described here is the essence of *Kalman filtering* (Kalman, 1960). Novel features of Kalman filters include:

- The theory is formulated in terms of state-space concepts, providing efficient utilization of the information contained in the input data.
- Estimation of the state is computed recursively; that is, each updated estimate of the state is computed from the previous estimate and the data currently available, so only the previous estimate requires storage.

In this section we present a brief review of Kalman filter theory¹⁰ to pave the way for the derivation of the decoupled extended Kalman filter described in the next section. The development of the theory usually begins with linear dynamical systems. To extend its use to nonlinear dynamical systems, a form of *linearization* is applied to the system; this latter part of the discussion is deferred to the next section.

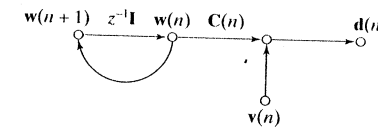


FIGURE 15.13 Signal-flow graph of linear, discrete-time dynamical system for describing the Kalman filter.

Consider then a *linear, discrete-time dynamical system* described by the signal flow graph shown in Fig. 15.13. The time-domain description of the system presented here is along similar lines to the state-space formalism presented in Section 15.3. In mathematical terms, Fig. 15.13 embodies the following pair of equations:

$$\mathbf{w}(n+1) = \mathbf{w}(n) \quad (15.59)$$

$$\mathbf{d}(n) = \mathbf{C}(n)\mathbf{w}(n) + \mathbf{v}(n) \quad (15.60)$$

The various quantities in the *process equation* (15.59) and the *measurement equation* (15.60) are described as follows:

- $\mathbf{w}(n)$ is the *state vector* of the system.
- $\mathbf{d}(n)$ is the *observation vector*.
- $\mathbf{C}(n)$ is the *measurement matrix*.
- $\mathbf{v}(n)$ is the *measurement noise*.

In the process equation (15.59) we have made two simplifying assumptions. First, the process equation is *noiseless*. Second, the transition matrix relating the states of the system at times $n+1$ and n is equal to the identity matrix. We have also used a new notation for the state in Fig. 15.13 for reasons that will become apparent in the next section.

The Kalman filtering problem may now be stated as follows:

Use the entire observed data, consisting of the set of vectors $\{\mathbf{d}(i)\}_{i=1}^n$, to find for each $n \geq 1$ the minimum mean-square error estimate of the state $\mathbf{w}(n)$.

Note that information on the state vector is not available. The problem is called *filtering* if $i = n$, *prediction* if $i > n$, and *smoothing* if $1 \leq i \leq n$. The solution to the problem is derived on the basis of the following assumptions (beyond the assumed linearity of the system):

1. The measurement noise $\mathbf{v}(n)$ is a zero mean, white noise process whose covariance matrix is defined by

$$E[\mathbf{v}(n)\mathbf{v}^T(k)] = \begin{cases} \mathbf{R}(n), & n = k \\ \mathbf{0}, & n \neq k \end{cases} \quad (15.61)$$

2. The initial value of the state, $\mathbf{w}(0)$, is uncorrelated with $\mathbf{v}(n)$ for all $n \geq 0$.

For an elegant derivation of the Kalman filter, we may use the notion of *innovations* (Kailath, 1968). Specifically, the *innovations process* associated with the observation vector $\mathbf{d}(n)$ is defined by

$$\boldsymbol{\alpha}(n) = \mathbf{d}(n) - \hat{\mathbf{d}}(n|n-1) \quad (15.62)$$

where $\hat{\mathbf{d}}(n|n-1)$ is the *minimum mean-square error estimate* of $\mathbf{d}(n)$, given all past values of the observation vector starting at time $n=1$ and extending up to time $n-1$.

By the “minimum mean-square error estimate” we mean that particular estimate that minimizes the mean-square error measured with respect to $\mathbf{d}(n)$. The innovations process $\alpha(n)$ may be regarded as a measure of the *new* information contained in $\mathbf{d}(n)$ that is not available in the predictable part $\hat{\mathbf{d}}(n|n-1)$. The innovations process $\alpha(n)$ has some nice properties as summarized here (Kailath, 1968):

1. The innovations process $\alpha(n)$ associated with $\mathbf{d}(n)$ is uncorrelated to all past observations $\mathbf{d}(1), \mathbf{d}(2), \dots, \mathbf{d}(n-1)$, as shown by

$$E[\alpha(n)\mathbf{d}^T(k)] = \mathbf{0} \quad \text{for } 1 \leq k \leq n-1$$

2. The innovations process consists of a sequence of random vectors that are uncorrelated with each other, as shown by

$$E[\alpha(n)\alpha^T(k)] = \mathbf{0} \quad \text{for } 1 \leq k \leq n-1$$

3. There is a one-to-one correspondence between the sequence of random vectors representing the observed data and the sequence of random vectors representing the innovations process, as shown by

$$\{\mathbf{d}(1), \mathbf{d}(2), \dots, \mathbf{d}(n)\} \rightleftharpoons \{\alpha(1), \alpha(2), \dots, \alpha(n)\} \quad (15.63)$$

We may now replace the correlated sequence of observed data with the uncorrelated (and therefore simpler) sequence of innovations *without any loss of information*. In so doing, the derivation of the Kalman filter is simplified by expressing the estimate of the state at time i , given the set of innovations $\{\alpha(k)\}_{k=1}^i$. In carrying out the analysis on this basis, we may derive the standard Kalman filter as summarized in Table 15.2. There are three new quantities in this algorithm that need to be defined:

- $\mathbf{K}(n, n-1)$ is the *error covariance matrix* defined by

$$\mathbf{K}(n, n-1) = E[\epsilon(n, n-1)\epsilon^T(n, n-1)] \quad (15.64)$$

where the *state error* $\epsilon(n, n-1)$ is itself defined by

$$\epsilon(n, n-1) = \mathbf{w}(n) - \hat{\mathbf{w}}(n|n-1) \quad (15.65)$$

where $\mathbf{w}(n)$ is the actual state and $\hat{\mathbf{w}}(n|n-1)$ is its one-step prediction based on past values of the observed data up to time $n-1$.

TABLE 15.2 Summary of the Kalman Filter

Compute for $n = 1, 2, 3, \dots$

$$\Gamma(n) = [\mathbf{C}(n)\mathbf{K}(n, n-1)\mathbf{C}^T(n) + \mathbf{R}(n)]^{-1}$$

$$\mathbf{G}(n) = \mathbf{K}(n, n-1)\mathbf{C}^T(n)\Gamma(n)$$

$$\alpha(n) = \mathbf{y}(n) - \mathbf{C}(n)\hat{\mathbf{w}}(n|n-1)$$

$$\hat{\mathbf{w}}(n+1|n) = \hat{\mathbf{w}}(n|n-1) + \mathbf{G}(n)\alpha(n)$$

$$\mathbf{K}(n+1, n) = \mathbf{K}(n, n-1) - \mathbf{G}(n)\mathbf{C}(n)\mathbf{K}(n, n-1)$$

- $\Gamma(n)$ is the *conversion factor* that relates the *filtered estimation error* $\mathbf{e}(n)$ to the innovations $\alpha(n)$ as shown by

$$\mathbf{e}(n) = \mathbf{R}(n)\Gamma(n)\alpha(n) \quad (15.66)$$

where

$$\mathbf{e}(n) = \mathbf{d}(n) - \hat{\mathbf{d}}(n|n) \quad (15.67)$$

where $\hat{\mathbf{d}}(n|n)$ is the estimate of the observation vector $\mathbf{d}(n)$ given all the observed data up to time n .

- $\mathbf{G}(n)$ is the *Kalman gain*, which determines the correction used to update the estimate of the state.

The type of Kalman filter summarized in Table 15.2 is designed to propagate the error covariance matrix $\mathbf{K}(n, n-1)$. This algorithm is therefore called the *covariance Kalman filtering algorithm*.

Square Root Kalman Filter

The covariance Kalman filter is prone to serious numerical difficulties. In particular the updated matrix $\mathbf{K}(n+1, n)$ is determined by the *Riccati equation*, which is defined by the last line of computation in Table 15.2. The right-hand side of the Riccati equation is the difference between two matrix quantities. Unless the numerical accuracy employed at every iteration of the algorithm is high enough, the updated matrix $\mathbf{K}(n+1, n)$ resulting from this computation may *not* be nonnegative definite. Such a solution is clearly unacceptable because $\mathbf{K}(n+1, n)$ represents a covariance matrix which by definition is nonnegative definite. The unstable behavior of the Kalman filter which results from numerical inaccuracies due to the use of finite word length arithmetic, is called the *divergence phenomenon*.

This problem may be overcome by propagating the square root of the error covariance matrix, $\mathbf{K}^{1/2}(n, n-1)$, rather than $\mathbf{K}(n, n-1)$ itself. Specifically, using the *Cholesky factorization*, we may express $\mathbf{K}(n, n-1)$ as follows (Golub and Van Loan, 1996):

$$\mathbf{K}(n, n-1) = \mathbf{K}^{1/2}(n, n-1)\mathbf{K}^{T/2}(n, n-1) \quad (15.68)$$

where $\mathbf{K}^{1/2}(n, n-1)$ is a lower triangular matrix, and $\mathbf{K}^{T/2}(n, n-1)$ is its transpose. In linear algebra, the Cholesky factor $\mathbf{K}^{1/2}(n, n-1)$ is commonly referred to as the *square root of $\mathbf{K}(n, n-1)$* . Thus a Kalman filter based on the Cholesky factorization is called a *square root Kalman filter*.¹¹ The important point is that the matrix product $\mathbf{K}^{1/2}(n, n-1)\mathbf{K}^{T/2}(n, n-1)$ is much less likely to become indefinite because the product of any square matrix and its transpose is always positive definite.

15.10 DECOUPLED EXTENDED KALMAN FILTER

Our primary interest in the Kalman filter is to exploit its unique properties to perform the supervised training of a recurrent network.¹² Given the architectural complexity of a recurrent network (e.g., recurrent multilayer perceptron), the important issue is how to proceed with this approach in a computationally feasible manner without compromising

the application of Kalman filter theory. The answer is found in using a *decoupled* form of the extended Kalman filter, in which the computational complexity is made to suit the requirements of a particular application and of available computational resources (Puskorius and Feldkamp, 1991).

Consider a recurrent network built around a static multilayer perceptron with W synaptic weights and p output nodes. Let the vector $\mathbf{w}(n)$ denote the synaptic weights of the entire network at time n . With adaptive filtering in mind, the state-space equations for the network may be modeled as follows (Singhal and Wu, 1989; Haykin, 1996):

$$\mathbf{w}(n+1) = \mathbf{w}(n) \quad (15.69)$$

$$\mathbf{d}_o(n) = \mathbf{c}(\mathbf{w}(n), \mathbf{u}(n), \mathbf{v}(n)) + \mathbf{v}(n) \quad (15.70)$$

where the weight vector $\mathbf{w}(n)$ plays the role of a state. The second argument $\mathbf{u}(n)$ and third argument $\mathbf{v}(n)$ pertaining to the vector valued function $\mathbf{c}(\cdot, \cdot, \cdot)$ denote the input vector and vector of recurrent node activities, respectively. In effect, Eq. (15.69) states that the model is residing in its "optimum" condition, with the transition matrix that takes the weight vector $\mathbf{w}(n)$ at time n and transforms it into $\mathbf{w}(n+1)$ at time $n+1$ being equal to the identity matrix. The optimum condition described here refers to a local or global minimum on the error surface of the recurrent network. The only source of nonlinearity in the model resides in the measurement equation (15.70). The vector $\mathbf{d}_o$ denotes the desired response of the model. With Eq. (15.70) representing the input-output equation of the model, it follows that $\mathbf{c}(\cdot, \cdot, \cdot)$ is the overall nonlinearity from the input layer to the output layer of the multilayer perceptron. The measurement noise vector $\mathbf{v}(n)$ in Eq. (15.70) is assumed to be a multivariate white noise process with zero mean and diagonal covariance matrix $\mathbf{R}(n)$.

It is important to note that in applying the extended Kalman filter to a recurrent network, there are two different contexts in which the term "state" is used:

- Evolution of the system via adaptive filtering, which manifests itself in changes to the recurrent network's weights through training; the vector $\mathbf{w}(n)$ takes care of this first notion of state.
- Operation of the recurrent network itself, exemplified by the recurrent node activities on which the function $\mathbf{c}$ depends; the vector $\mathbf{v}(n)$ takes care of this second notion of state.

By comparing the model described in Eqs. (15.69) and (15.70) with the linear dynamical model of Eqs. (15.59) and (15.60), we see that the only difference between these two models is in the nonlinear form of the measurement equation. To prepare the way for the application of Kalman filter theory to the state-space model just described, we must therefore first linearize Eq. (15.70) and recast it in the form

$$\mathbf{d}(n) = \mathbf{C}(n)\mathbf{w}(n) + \mathbf{v}(n) \quad (15.71)$$

where $\mathbf{C}(n)$ is the p -by- W measurement matrix of the linearized model, and we have used $\mathbf{d}(n)$ to distinguish it from $\mathbf{d}_o(n)$ in Eq. (15.70). The linearization consists of the partial derivatives of the p outputs of the whole network with respect to the W weights of the model as shown by

$$\mathbf{C}(n) = \begin{bmatrix} \frac{\partial c_1}{\partial w_1} & \frac{\partial c_1}{\partial w_2} & \dots & \frac{\partial c_1}{\partial w_W} \\ \frac{\partial c_2}{\partial w_1} & \frac{\partial c_2}{\partial w_2} & \dots & \frac{\partial c_2}{\partial w_W} \\ \vdots & \vdots & & \vdots \\ \frac{\partial c_p}{\partial w_1} & \frac{\partial c_p}{\partial w_2} & \dots & \frac{\partial c_p}{\partial w_W} \end{bmatrix} \quad (15.72)$$

where $c_i, i = 1, 2, \dots, p$ denotes the i th element of the nonlinearity $\mathbf{c}(\mathbf{w}(n), \mathbf{u}(n), \mathbf{v}(n))$. The partial derivatives in Eq. (15.72) are evaluated at $\mathbf{w}(n) = \hat{\mathbf{w}}(n)$, where $\hat{\mathbf{w}}(n)$ is the estimate of the weight vector $\mathbf{w}(n)$ computed by the extended Kalman filter at time n given the observed data up to time $n-1$ (Haykin, 1996). In practice, these partial derivatives are computed by using the back-propagation-through-time or real-time recurrent learning algorithm. In effect, the extended Kalman filter builds on one of the other of these two algorithms described in Sections 15.7 and 15.8. This implies that $\mathbf{c}$ must be a function of the recurrent node activities as stated. In fact, for a single layer recurrent network the matrix $\mathbf{C}(n)$ can be composed from the elements of the matrices $\mathbf{A}_j(n)$ as computed by the RTRL algorithm in Eq. (15.52). Thus the measurement matrix $\mathbf{C}(n)$ is the dynamic derivative matrix of the network outputs with respect to the network's free parameters. Just as the recurrent node activities of the network at time step $(n+1)$ are a function of the corresponding values from the previous time step n , in an analogous manner we find that the derivatives of recurrent node activities with respect to the network's free parameters at time step $(n+1)$ are a function of the corresponding values from the previous time step n as expressed in the RTRL equations.

Suppose now the synaptic weights of the network are partitioned into g groups with group i containing k_i neurons, for example. The measurement matrix $\mathbf{C}$ defined in Eq. (15.72) is the p -by- W matrix of derivatives of network outputs with respect to all the weights in the network. The dependence of matrix $\mathbf{C}(n)$ on the input vector $\mathbf{u}(n)$ is implicitly defined in Eq. (15.72). The matrix $\mathbf{C}(n)$ thus defined contains all the derivatives that are necessary for any decoupled version of the extended Kalman filter. For example, if the *global extended Kalman filter* (GEKF) is used (i.e., we have no decoupling), $g = 1$ and the whole $\mathbf{C}(n)$ is as defined in Eq. (15.72). On the other hand, if the *decoupled extended Kalman filter* (DEKF) is used, then the "global" measurement matrix $\mathbf{C}(n)$ must be arranged so that the weights corresponding to a given neuron in the network are grouped as a single block within $\mathbf{C}(n)$, where each block is identified by index $i = 1, 2, \dots, g$. In the latter case, the matrix $\mathbf{C}(n)$ is merely the *concatenation* of the individual $\mathbf{C}_i$'s, as shown here

$$\mathbf{C}(n) = [\mathbf{C}_1(n), \mathbf{C}_2(n), \dots, \mathbf{C}_g(n)]$$

In any event, regardless of the level of decoupling employed, the entire matrix $\mathbf{C}(n)$ must be computed as defined in Eq. (15.72).

The stage is now set for the application of the Kalman filtering algorithm summarized in Table 15.2. In particular, for the linearized dynamic model described by Eqs. (15.69) and (15.71), we have (Puskorius and Feldkamp, 1991):

$$\Gamma(n) = \left[\sum_{i=1}^g \mathbf{C}_i(n) \mathbf{K}_i(n, n-1) \mathbf{C}_i^T(n) + \mathbf{R}(n) \right]^{-1} \quad (15.73)$$

$$\mathbf{G}_i(n) = \mathbf{K}_i(n, n-1) \mathbf{C}_i^T(n) \Gamma(n) \quad (15.74)$$

$$\boldsymbol{\alpha}(n) = \mathbf{d}(n) - \hat{\mathbf{d}}(n|n-1) \quad (15.75)$$

$$\mathbf{w}_i(n+1|n) = \hat{\mathbf{w}}_i(n|n-1) + \mathbf{G}_i(n) \boldsymbol{\alpha}(n) \quad (15.76)$$

$$\mathbf{K}_i(n+1, n) = \mathbf{K}_i(n, n-1) - \mathbf{G}_i(n) \mathbf{C}_i(n) \mathbf{K}_i(n, n-1) \quad (15.77)$$

where $i = 1, 2, \dots, g$. The parameter vectors and signal vectors in Eq. (15.73) to (15.77) are described as follows:

$\Gamma(n)$ = p -by- p matrix, denoting the global conversion factor for the entire network

$\mathbf{G}_i(n)$ = W_i -by- p matrix, denoting the Kalman gain for group i of neurons

$\boldsymbol{\alpha}(n)$ = p -by-1 vector, denoting the innovations defined as the difference between the desired response $\mathbf{d}(n)$ for the linearized system and its estimate $\hat{\mathbf{d}}(n|n-1)$ based on input data available at time $n-1$; the estimate $\hat{\mathbf{d}}(n|n-1)$ is represented by the actual output vector $\mathbf{y}(n)$ of the network residing in state $\{\hat{\mathbf{w}}_i(n|n-1)\}$, which is produced in response to the input $\mathbf{u}(n)$

$\hat{\mathbf{w}}_i(n|n-1)$ = W -by-1 vector, denoting the estimate of the weight vector $\mathbf{w}_i(n)$ for group i at time n , given the observed data up to time $n-1$

$\mathbf{K}_i(n, n-1)$ = k_i -by- k_i matrix, denoting the error covariance matrix for group i of neurons

The summation included in the definition of the global conversion factor $\Gamma(n)$ in Eq. (15.73) accounts for the decoupled nature of the extended Kalman filter.

It is important to understand that in the DEKF algorithm the decoupling really determines which particular elements of the global error covariance matrix $\mathbf{K}(n, n-1)$ are to be maintained and updated. In fact, all computational savings are due to ignoring the maintenance and updates associated with those off-diagonal blocks of the global error covariance matrix $\mathbf{K}(n, n-1)$, which would otherwise correspond to coupling of different groups of synaptic weights.

The DEKF algorithm encoded by Eqs. (15.73) to (15.77) minimizes the cost function:

$$\mathcal{E}(n) = \frac{1}{2} \sum_{j=1}^n \|\mathbf{e}(j)\|^2 \quad (15.78)$$

where $\mathbf{e}(j)$ is the error vector defined by

$$\mathbf{e}(j) = \mathbf{d}(j) - \mathbf{y}(j), \quad j = 1, 2, \dots, n$$

where $\mathbf{y}(j)$ is the actual output of the network using all available information up to and including time j . Note that, in general, $\mathbf{e}(j) \neq \boldsymbol{\alpha}(j)$.

Artificial Process Noise

The nonlinear dynamical system modeled in Eqs. (15.69) and (15.70) is *unforced*, i.e., that the process equation (15.69) has no external inputs. This deficiency can lead to serious numerical difficulties, and therefore the divergence of the Kalman filter when it operates in a finite precision environment. As explained in Section 15.9, the divergence phenomenon may be overcome through the use of square root filtering.

Another way of circumventing the divergence phenomenon is to use a heuristic mechanism that involves artificially adding *process noise* to the process equation, as shown by

$$\mathbf{w}_i(n+1) = \mathbf{w}_i(n) + \boldsymbol{\omega}_i(n), \quad i = 1, 2, \dots, g \quad (15.79)$$

where $\boldsymbol{\omega}_i(n)$ is the process noise. It is assumed that $\boldsymbol{\omega}_i(n)$ is a multivariate white noise with zero mean and diagonal covariance matrix $\mathbf{Q}_i(n)$. The artificially added process noise $\boldsymbol{\omega}_i(n)$ is naturally independent of both the measurement noise $\mathbf{v}(n)$ and the initial state of the network. The net effect of adding $\boldsymbol{\omega}_i(n)$ to the process equation (15.79) is to modify the Riccati equation for updating the error covariance matrix as follows (Haykin, 1996):

$$\mathbf{K}_i(n+1, n) = \mathbf{K}_i(n, n-1) - \mathbf{G}_i(n) \mathbf{C}_i(n) \mathbf{K}_i(n, n-1) + \mathbf{Q}_i(n) \quad (15.80)$$

Provided that $\mathbf{Q}_i(n)$ is large enough for all i , then $\mathbf{K}_i(n+1, n)$ is assured of remaining nonnegative definite for all n .

In addition to overcoming these numerical difficulties, the artificial insertion of process noise $\boldsymbol{\omega}_i(n)$ into the process equation has the following beneficial effect: There is less likelihood for the algorithm to be trapped at a local minimum during the training process. This in turn results in a significant improvement in training performance in terms of rate of convergence and quality of solution (Puskorius and Feldkamp, 1991).

Summary of the DEKF Algorithm

Table 15.3 presents a summary of the DEKF algorithm based on Eqs. (15.73) to (15.76) and Eq. (15.80). This table also includes details of initialization of the algorithm.

A final comment on the extended Kalman filter is in order. The DEKF algorithm summarized in Table 15.3 refers to an entire family of possible *information-preserving learning procedures*, including the GEKF. As a general rule, we expect the DEKF to produce a performance, in terms of solution quality, that approaches the GEKF but is not expected to surpass it. On the other hand, the DEKF is always computationally less demanding than the GEKF. Notwithstanding this computational advantage, current computer speeds and memory sizes have now made GEKF feasible for some practical problems, especially in off-line training of recurrent networks.

TABLE 15.3 Summary of the DEKF Algorithm*Initialization:*

1. Set the synaptic weights of the recurrent network to small values selected from a uniform distribution.
2. Set the diagonal elements of the covariance matrix $\mathbf{Q}(n)$ (characterizing the artificially inserted process noise $\omega(n)$) equal to 10^{-6} to 10^{-2} .
3. Set $\mathbf{K}(1, 0) = \delta^{-1} \mathbf{I}$, $\delta =$ small positive constant.

Computations:

For $n = 1, 2, \dots$, compute

$$\Gamma(n) = \left[\sum_{i=1}^g \mathbf{C}_i(n) \mathbf{K}_i(n, n-1) \mathbf{C}_i^T(n) + \mathbf{R}(n) \right]^{-1}$$

$$\mathbf{G}_i(n) = \mathbf{K}_i(n, n-1) \mathbf{C}_i^T(n) \Gamma(n)$$

$$\alpha(n) = \mathbf{d}(n) - \hat{\mathbf{d}}(n|n-1)$$

$$\hat{\mathbf{w}}_i(n+1|n) = \hat{\mathbf{w}}_i(n|n-1) + \mathbf{G}_i(n) \alpha(n)$$

$$\mathbf{K}_i(n+1, n) = \mathbf{K}_i(n, n-1) - \mathbf{G}_i(n) \mathbf{C}_i(n) \mathbf{K}_i(n, n-1) + \mathbf{Q}_i(n)$$

where in the third line, $\hat{\mathbf{d}}(n|n-1)$ is the actual output vector $\mathbf{y}(n)$ of the network produced in response to the input vector $\mathbf{u}(n)$.

Note: For $g = 1$ (i.e., no decoupling), the DEKF algorithm becomes the global extended Kalman filtering (GEKF) algorithm.

Computational Complexity

Table 15.4 presents a comparison of the computational complexity of the three learning algorithms discussed in this chapter: back-propagation through time, real-time recurrent learning, and decoupled extended Kalman filter. The computational complexity of these algorithms increases in the order arranged here.

11 COMPUTER EXPERIMENT

In this experiment we revisit the simulation of the nonlinear time series studied in Section 13.5. The time series is defined by the frequency-modulated signal:

$$x(n) = \sin(n + \sin(n^2)) \quad n = 0, 1, 2, \dots$$

We will investigate two different structures for the simulation:

- Recurrent multilayer perceptron (RMLP) consisting of 1 input node, first hidden layer of 10 recurrent neurons, second hidden layer of 10 neurons, and 1 linear output neuron.
- Focused time lagged feedforward network (TLFN), consisting of a tapped-delay-time memory with 20 taps and a multilayer perceptron with 10 hidden neurons and 1 linear output neuron.

The RMLP has slightly more synaptic weights than the focused TLFN, but half the memory (10 recurrent nodes versus 20 taps).

TABLE 15.4 Comparison of the Computational Complexity of Learning Algorithms for Recurrent Networks

S = number of states

W = number of synaptic weights

L = length of training sequence

1. Back-propagation-through-time (BPTT) algorithm:
 - Time, storage space requirements: $O(WL + SL)$, $O(WL + SL)$
2. Real-time recurrent learning (RTRL) algorithm:
 - Time, storage space requirements: $O(WS^2L)$, $O(WS)$
3. Decoupled extended Kalman filtering (DEKF) algorithm:
 - At the minimum, DEKF incurs the same expense (in both time and space) for computing derivatives via RTRL or BPTT; for BPTT the time and space requirements are scaled by p , the number of network outputs, relative to the standard BPTT for which derivatives of a single scalar error term are computed.
 - In addition, DEKF requires time complexity $O(p^2W + p \sum_{i=1}^g k_i^2)$ and storage space $O(\sum_{i=1}^g k_i^2)$, where g is the number of groups and k_i is the number of neurons in group i . In the limit of a single weight group as in GEKF, these requirements become time and space storage: $O(pW^2)$ and $O(W^2)$, respectively.

The RMLP was trained using the DEKF algorithm. The TLFN was trained using two versions of the extended Kalman filter: (1) the GEKF algorithm (i.e., global version), and (2) the DEKF algorithm (i.e., decoupled version). The details of these two algorithms are:

• **GEKF:**

δ = parameter used to initialize the error covariance matrix $\mathbf{K}(n, n-1)$
= 0.01

$\mathbf{R}(n)$ = covariance matrix of measurement noise $\mathbf{v}(n)$: $\mathbf{R}(0) = 100$ at the start of training and then annealed to $\mathbf{R}(n) = 3$ at the end of training

$\mathbf{Q}(n)$ = covariance matrix of artificial process noise $\omega(n)$: $\mathbf{Q}(0) = 10^{-2}$ at the start of training and then annealed to $\mathbf{Q}(n) = 10^{-6}$ at the end of training

The annealing of $\mathbf{R}(n)$ and $\mathbf{Q}(n)$ has the effect of accelerating the learning rate as the training progresses.

• **DEKF:**

g = number of groups

$$= \begin{cases} 21 & \text{for the RMLP} \\ 11 & \text{for the focused TLFN} \end{cases}$$

All other parameters are the same as those used for the GEKF.

The training was performed on a sequence of 4000 samples. For the RMLP, subsets of length 100 were used, with the processing of 30,000 subsets over the entire training run. Each data point in the training set of 4000 samples was processed approximately 750 times. For the focused TLFN, each data point in the training set was also processed about 750 times. In both cases the testing was performed on 300 data points.

Figure 15.14 presents the one-step predicted waveform $\hat{y}(n)$ computed by the RMLP trained on the DEKF algorithm. This figure also includes the actual waveform

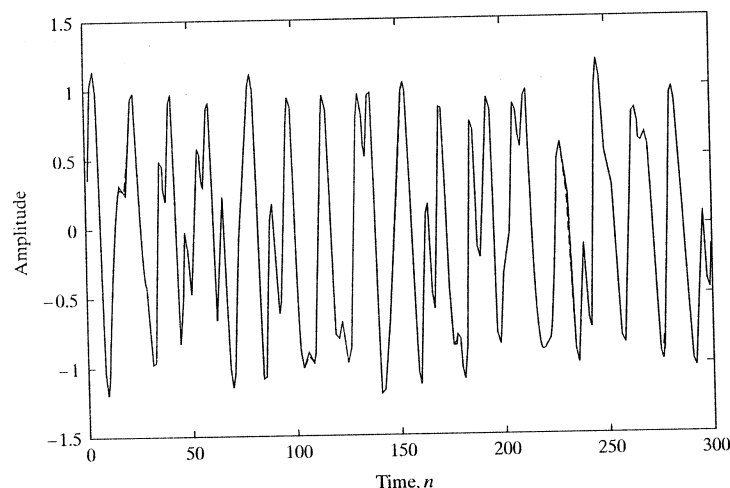


FIGURE 15.14 Superposition of actual waveform (solid) and predicted waveform (dashed) for the computer experiment on modeling; the predicted waveform was computed using the RMLP trained on the DEKF algorithm.

$y(n)$. These two waveforms are hardly distinguishable from each other. Figure 15.15a shows the prediction error

$$e(n) = y(n) - \hat{y}(n)$$

produced by the RMLP. The corresponding prediction errors produced by the focused TLFN trained on the GEKF and DEKF algorithms are shown in Figs. 15.15b and 15.15c, respectively. By comparing the results presented in Fig. 15.15 among themselves and also against the simulation results reported in Section 13.5, we may make the following observations:

1. The most accurate simulation in a mean-square error sense was produced by the RMLP trained on the DEKF algorithm; the variance of the prediction error was 1.1839×10^{-4} , computed over 5980 samples.
2. For the focused TLFN, the most accurate simulation in a mean-square error sense was produced by using GEKF training. For GEKF training the variance of the prediction error was 1.3351×10^{-4} , whereas for DEKF training it was 1.5871×10^{-4} . Both computations were again made using 5980 samples.
3. For the focused TLFN trained on the standard back-propagation algorithm, the variance of the prediction error reported in Section 13.5 was 1.2×10^{-3} , an order of magnitude worse than that obtained with the GEKF and DEKF algorithms.

The superior learning performance of the extended Kalman filter over back-propagation is due to its information-preserving property.

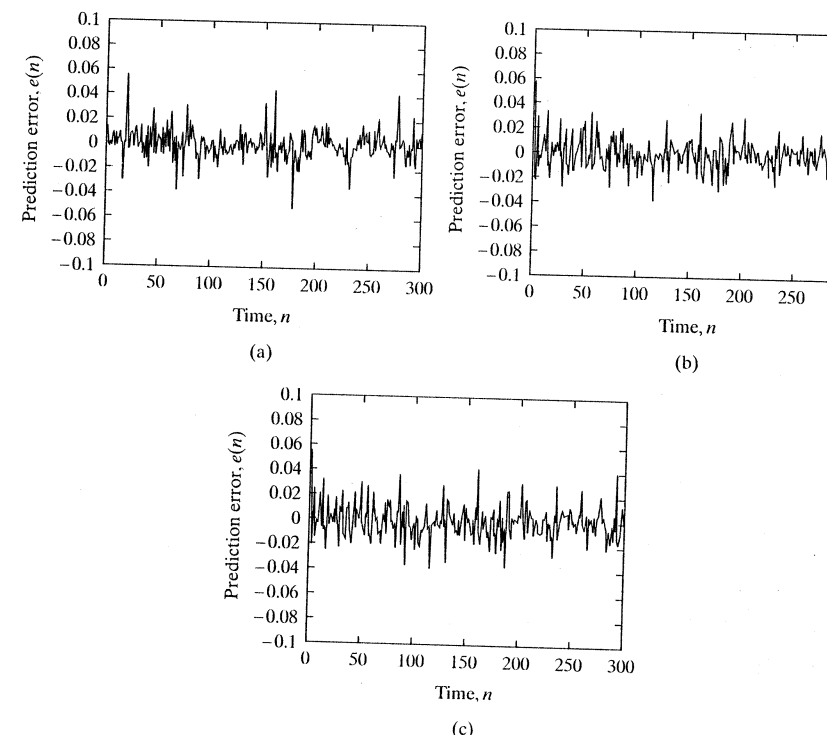


FIGURE 15.15 Prediction error waveforms for three different simulations: (a) RMLP with DEKF training, error variance = 1.1839×10^{-4} . (b) Focused TLFN with GEKF training error variance = 1.3351×10^{-4} . (c) Focused TLFN with DEKF training, error variance = 1.5871×10^{-4} .

15.12 VANISHING GRADIENTS IN RECURRENT NETWORKS

A problem that may require attention in practical applications of a recurrent network is the *vanishing gradients problem*, which pertains to the training of the network to produce a desired response at the current time that depends on input data in the distant past (Hochreiter, 1991; Bengio et al., 1994). The point is that because of the combined nonlinearities, an infinitesimal change of a temporally distant input may have almost no effect on network training. The problem may arise even if a large change in the temporally distant input has an effect, but the effect is not measurable by the gradient. This vanishing gradients problem makes the learning of long-term dependencies in gradient-based training algorithms difficult if not virtually impossible in certain cases.

In Bengio et al. (1994), it is argued that for many practical applications it is necessary that a recurrent network be able to store state information for an arbitrary duration

and to do so in the presence of noise. The long-term storage of definite bits of information in the state variables of the recurrent network is referred to as *information latching*. The information latching must be *robust* so that the stored state information cannot be easily deleted by events that are unrelated to the learning task at hand. In specific terms, we may state the following (Bengio et al., 1994):

Robust information latching in a recurrent network is accomplished if the states of the network are contained in the reduced attracting set of a hyperbolic attractor.

The notion of a hyperbolic attractor was discussed in Chapter 14. The *reduced attracting set* of a hyperbolic attractor is the set of points in the basin of attraction for which all the eigenvalues of the associated Jacobian have an absolute value less than 1. The implication here is that if a state $\mathbf{x}(n)$ of the recurrent network is in the basin of attraction of a hyperbolic attractor but not in the reduced attracting set, then the size of a ball of uncertainty around $\mathbf{x}(n)$ will grow exponentially with increasing time n , as illustrated in Fig. 15.16a. Therefore, small perturbations (noise) in the input applied to the recurrent network could push the trajectory toward another (possibly wrong) basin of attraction. If, however, the state $\mathbf{x}(n)$ remains in the reduced attracting set of the hyperbolic attractor, a bound on the input can be found that guarantees $\mathbf{x}(n)$ to remain within a certain distance of the attractor, as illustrated in Fig. 15.16b.

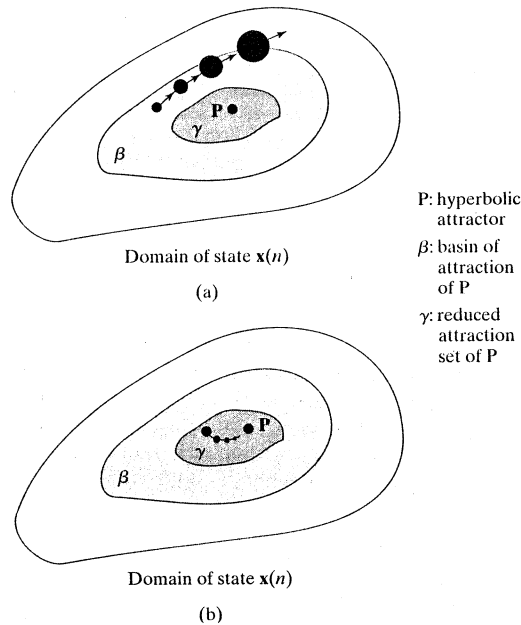


FIGURE 15.16 (a) State $\mathbf{x}(n)$ resides in the basin of attraction β but outside the reduced attraction set γ . (b) State $\mathbf{x}(n)$ resides inside the reduced attraction set γ .

Long-Term Dependencies

To appreciate the impact of robust information latching on gradient-based learning, note that the adjustment applied to the weight vector $\mathbf{w}$ of a recurrent network at time n is defined by

$$\Delta \mathbf{w}(n) = -\eta \frac{\partial \mathcal{E}_{\text{total}}}{\partial \mathbf{w}}$$

where η is the learning-rate parameter and $\partial \mathcal{E}_{\text{total}} / \partial \mathbf{w}$ is the gradient of the cost function $\mathcal{E}_{\text{total}}$ with respect to $\mathbf{w}$. The cost function $\mathcal{E}_{\text{total}}$ is typically defined by

$$\mathcal{E}_{\text{total}} = \frac{1}{2} \sum_i \|\mathbf{d}_i(n) - \mathbf{y}_i(n)\|^2$$

where $\mathbf{d}_i(n)$ is the desired response and $\mathbf{y}_i(n)$ is the actual response of the network at time n for the i th pattern. Hence, we may write

$$\begin{aligned} \Delta \mathbf{w}(n) &= \eta \sum_i \left(\frac{\partial \mathbf{y}_i(n)}{\partial \mathbf{w}} \right) (\mathbf{d}_i(n) - \mathbf{y}_i(n)) \\ &= \eta \sum_i \left(\frac{\partial \mathbf{y}_i(n)}{\partial \mathbf{x}_i(n)} \frac{\partial \mathbf{x}_i(n)}{\partial \mathbf{w}} \right) (\mathbf{d}_i(n) - \mathbf{y}_i(n)) \end{aligned} \quad (15.81)$$

where in the second line we have used the chain rule of calculus; the state vector $\mathbf{x}_i(n)$ pertains to the i th pattern (example) in the training sample. In applying algorithms such as back-propagation through time, the partial derivatives of the cost function are computed with respect to independent weights at different time indices. We may expand on the result in Eq. (15.81) by writing

$$\Delta \mathbf{w}(n) = \eta \sum_i \left(\frac{\partial \mathbf{y}_i(n)}{\partial \mathbf{x}_i(n)} \sum_{k=1}^n \frac{\partial \mathbf{x}_i(n)}{\partial \mathbf{w}(k)} \right) (\mathbf{d}_i(n) - \mathbf{y}_i(n))$$

Applying the chain rule of calculus a second time yields

$$\Delta \mathbf{w}(n) = \eta \sum_i \left(\frac{\partial \mathbf{y}_i(n)}{\partial \mathbf{x}_i(n)} \sum_{k=1}^n \frac{\partial \mathbf{x}_i(n)}{\partial \mathbf{x}_i(k)} \frac{\partial \mathbf{x}_i(k)}{\partial \mathbf{w}(k)} \right) (\mathbf{d}_i(n) - \mathbf{y}_i(n)) \quad (15.82)$$

We now recognize that in light of the state equation (15.2), we have

$$\mathbf{x}_i(n) = \boldsymbol{\varphi}(\mathbf{x}_i(k), \mathbf{u}(n)), \quad 1 \leq k < n$$

Hence, we may interpret $\partial \mathbf{x}_i(n) / \partial \mathbf{x}_i(k)$ as the Jacobian of the nonlinear function $\boldsymbol{\varphi}(\cdot, \cdot)$ expanded over $n - k$ time steps, as shown by

$$\begin{aligned} \frac{\partial \mathbf{x}_i(n)}{\partial \mathbf{x}_i(k)} &= \frac{\partial \boldsymbol{\varphi}(\mathbf{x}_i(k), \mathbf{u}(n))}{\partial \mathbf{x}_i(k)} \\ &= \mathbf{J}_{\mathbf{x}}(n, n - k) \end{aligned} \quad (15.83)$$

In Bengio et al. (1994), it is shown that if the input $\mathbf{u}(n)$ is such that the recurrent network remains robustly latched to a hyperbolic attractor after time $n = 0$, then the Jacobian $\mathbf{J}_x(n, k)$ is an exponentially decreasing function of k so that

$$\det(\mathbf{J}_x(n, k)) \rightarrow 0 \quad \text{as } k \rightarrow \infty \quad \text{for all } n \quad (15.84)$$

The implication of Eq. (15.84) is that a *small* change in the weight vector $\mathbf{w}$ of the network is experienced mostly in the near past (i.e., values of k close to the current time n). There may exist an adjustment $\Delta \mathbf{w}$ to the weight vector $\mathbf{w}$ at time n that would permit the current state $\mathbf{x}(n)$ to move to another possibly better basin of attraction, but the gradient of the cost function $\mathcal{E}_{\text{total}}$ with respect to $\mathbf{w}$ does not carry that information.

To conclude, assuming that hyperbolic attractors are used to store state information in a recurrent network by means of gradient-based learning, we find that either

- the network is *not* robust to the presence of noise in the input signal or
- the network is unable to discover *long-term dependencies* (i.e., relationships between target outputs and inputs that occur in the distant past).

Possible procedures for alleviating the difficulties that arise due to vanishing gradients in recurrent networks include the following:¹³

- Increased temporal span of input–output dependencies by presenting the network, during training, with the shortest strings of symbols first; see the heuristics presented in Section 15.6
- Use of the extended Kalman filter or its decoupled version for a more efficient use of available information than gradient-based learning algorithms; the extended Kalman filter is discussed in Section 15.10
- Use of elaborate optimization methods such as pseudo-Newton and simulated annealing (Bengio et al., 1994); second-order optimization methods and simulated annealing are described in Chapters 4 and 11, respectively

13 SYSTEM IDENTIFICATION

System identification is the experimental approach to the modeling of a process or a plant of unknown parameters.¹⁴ It involves the following steps: experimental planning, the selection of a model structure, parameter estimation, and model validation. The procedure of system identification, as pursued in practice, is iterative in nature in that we may have to go back and forth between these steps until a satisfactory model is built.

Suppose then we have an unknown nonlinear dynamical plant, and the requirement is to build a suitably parameterized identification model for it. We have the choice of basing the identification procedure on a state-space model or an input–output model. The decision as to which of these two representations is used hinges on prior information of the inputs and observables of the system. In what follows, both representations are discussed.

System Identification Using the State-Space Model

Suppose that the given plant is described by the state-space model:

$$\mathbf{x}(n+1) = \mathbf{f}(\mathbf{x}(n), \mathbf{u}(n)) \quad (15.85)$$

$$\mathbf{y}(n) = \mathbf{h}(\mathbf{x}(n)) \quad (15.86)$$

where $\mathbf{f}(\cdot, \cdot)$ and $\mathbf{h}(\cdot)$ are vector-valued nonlinear functions, both of which are assumed to be unknown; Eq. (15.86) is a generalization of Eq. (15.11). We use two neural networks to identify the system, one for dealing with the process equation (15.85) and the other for dealing with the measurement equation (15.86), as depicted in Fig. 15.17.

We recognize that the state $\mathbf{x}(n)$ is the one-step delayed version of $\mathbf{x}(n+1)$. Let $\hat{\mathbf{x}}(n+1)$ denote the estimate of $\mathbf{x}(n+1)$ produced by the first neural network, labeled network I in Fig. 15.17a. This network operates on a concatenated input consisting of the external input $\mathbf{u}(n)$ and the state $\mathbf{x}(n)$ to produce $\hat{\mathbf{x}}(n+1)$. The estimate $\hat{\mathbf{x}}(n+1)$ is subtracted from the actual state $\mathbf{x}(n+1)$ to produce the error vector

$$\mathbf{e}_I(n+1) = \mathbf{x}(n+1) - \hat{\mathbf{x}}(n+1)$$

where $\mathbf{x}(n+1)$ plays the role of desired response. It is assumed that the actual state $\mathbf{x}(n)$ is physically accessible for it to be used in this way. The error vector $\mathbf{e}_I(n+1)$ is in

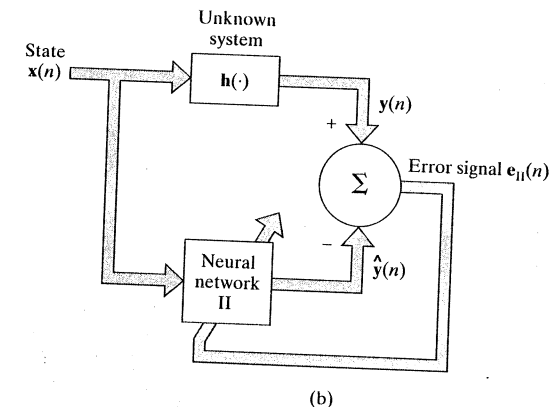
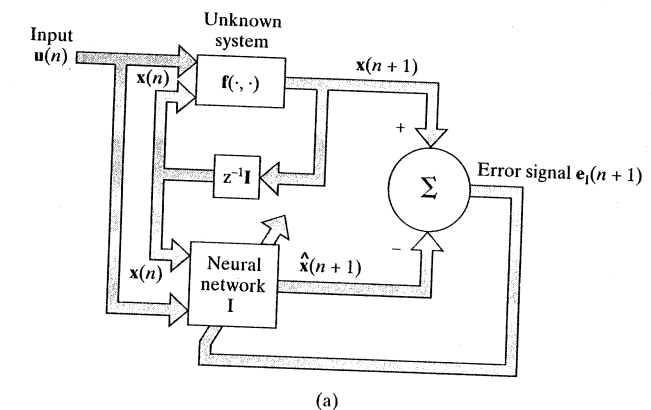


FIGURE 15.17 State-space solution for the system identification problem.

turn used to adjust the synaptic weights of neural network I, as indicated in Fig. 15.17a, so as to minimize the cost function based on the error vector $\mathbf{e}_I(n+1)$ in some statistical sense.

The second neural network, labeled network II in Fig. 15.17b, operates on the actual state $\mathbf{x}(n)$ of the unknown plant to produce an estimate $\hat{\mathbf{y}}(n)$ of the actual output $\mathbf{y}(n)$. The estimate $\hat{\mathbf{y}}(n)$ is subtracted from $\mathbf{y}(n)$ to produce the second error vector

$$\mathbf{e}_{II}(n) = \mathbf{y}(n) - \hat{\mathbf{y}}(n)$$

where $\mathbf{y}(n)$ plays the role of desired response. The error vector $\mathbf{e}_{II}(n)$ is then used to adjust the synaptic weights of network II to minimize the Euclidean norm of the error vector $\mathbf{e}_{II}(n)$ in some statistical sense.

The two neural networks shown in Fig. 15.17 operate in a synchronous fashion to provide a state-space solution to the system identification problem (Narendra and Parthasarathy, 1990). Such a model is referred to as a *series-parallel identification model* in recognition of the fact that the actual state of the unknown system (rather than that of the identification model) is fed into the identification model, as depicted in Fig. 15.17a. In light of the discussion presented at the end of Section 15.9, this form of training is an example of teacher forcing.

The series-parallel identification model of Fig. 15.17a should be contrasted against a *parallel identification model* where the $\mathbf{x}(n)$ applied to the neural network I is replaced with $\hat{\mathbf{x}}(n)$; the $\hat{\mathbf{x}}(n)$ is derived from the network's own output $\hat{\mathbf{x}}(n+1)$ by passing it through a unit delay z^{-1} . The practical benefit of this alternative model of training is that the neural network model is operated in exactly the same way as the unknown system, that is, the way in which the model will be used after the training is completed. It is therefore likely that the model developed via the parallel training mode may exhibit an autonomous behavior that is superior to the autonomous behavior of the network model developed via the series-parallel training mode. The disadvantage of the parallel training mode, however, is that it may take longer than the series-parallel training mode; see the discussion on teacher forcing in Section 15.9. Specifically, in our present situation, the estimate $\hat{\mathbf{x}}(n)$ of the state used in the parallel training model is ordinarily not as accurate as the actual state $\mathbf{x}(n)$ used in the series-parallel training mode.

Input-Output Model

Suppose next that the unknown plant is only accessible through its output. To simplify the presentation, let the system be of a single input, single output kind. Let $y(n)$ denote the output of the system due to the input $u(n)$ for varying discrete-time n . Then, choosing to work with the NARX model, the identification model takes the form:

$$\hat{y}(n+1) = \varphi(y(n), \dots, y(n-q+1), u(n), \dots, u(n-q+1))$$

where q is the order of the unknown system. At time $n+1$, the q past values of the input and the q past values of the output are all available. The model output $\hat{y}(n+1)$ represents an estimate of the actual output $y(n+1)$. The estimate $\hat{y}(n+1)$ is subtracted from $y(n+1)$ to produce the error signal

$$e(n+1) = y(n+1) - \hat{y}(n+1)$$

where $y(n+1)$ plays the role of desired response. The error $e(n+1)$ is used to adjust the synaptic weights of the neural network so as to minimize the error in some statistical sense. The identification model of Fig. 15.18 is of a series-parallel form (i.e., teacher forcing form) because the actual output of the system (rather than that of the identification model) is fed back to the input of the model.

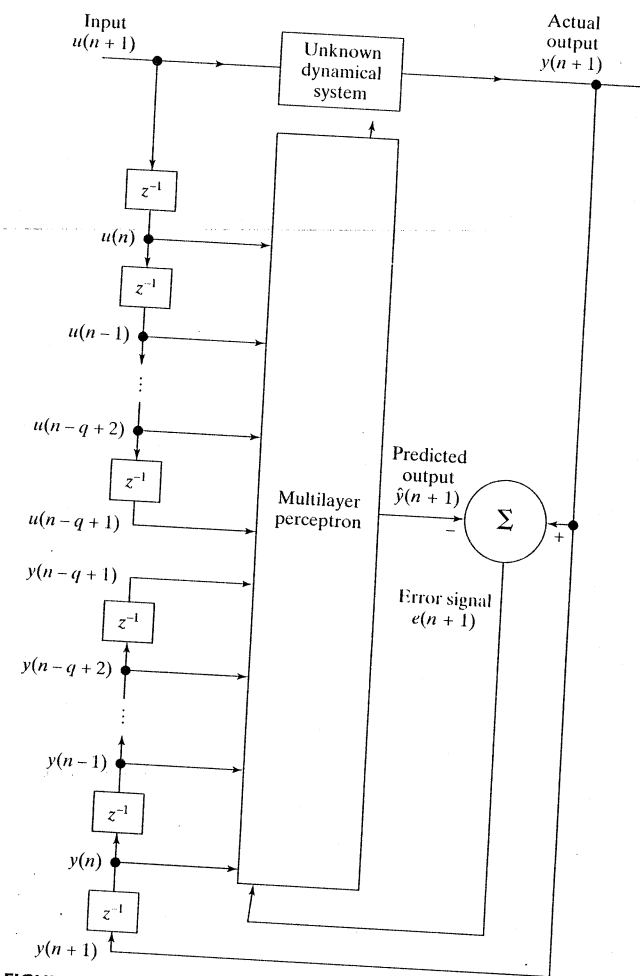


FIGURE 15.18 NARX solution for the system identification problem.

14 MODEL REFERENCE ADAPTIVE CONTROL

Another important application of recurrent networks is in the design of *feedback control systems* where the states of a plant are coupled nonlinearly with imposed controls (Puskorius and Feldkamp, 1994; Puskorius et al., 1996). The design of the system is further complicated by other factors such as the presence of unmeasured and random disturbances, the possibility of a nonunique plant inverse, and the presence of plant states that are unobservable.

A control strategy well suited for the use of neural networks is the *model reference adaptive control* (MRAC),¹⁵ where the implicit assumption is that the designer is sufficiently familiar with the system under consideration (Narendra and Annaswamy, 1989). Figure 15.19 shows the block diagram of such a system, where adaptivity is used to account for the fact that the dynamics of the plant are unknown. The controller and the plant form a closed loop feedback system, thereby constituting an *externally recurrent network*. The plant receives an input $u_c(n)$ from the controller along with an external disturbance $u_d(n)$. Accordingly, the plant evolves in time as a function of the imposed inputs and the plant's own state $x_p(n)$. The output of the plant, denoted by $y_p(n+1)$, is a function of $x_p(n)$. The plant output may also be corrupted by measurement noise.

The controller receives two inputs: an externally specified reference signal $r(n)$, and $y_p(n)$ representing a one-step delayed version of the plant output $y_p(n+1)$. The controller produces a vector of control signals defined by

$$u_c(n) = f_1(x_c(n), y_p(n), r(n), w)$$

where $x_c(n)$ is the controller's own state and w is a parameter vector that is available for adjustment. The vector-valued function $f_1(\cdot, \cdot, \cdot, \cdot)$ defines the input-output behavior of the controller.

The desired response $d(n+1)$ for the plant is supplied by the output of a stable *reference model*, which is produced in response to the reference $r(n)$. The desired response $d(n+1)$ is therefore a function of the reference signal $r(n)$ and the reference model's own state $x_r(n)$, as shown by

$$d(n+1) = f_2(x_r(n), r(n))$$

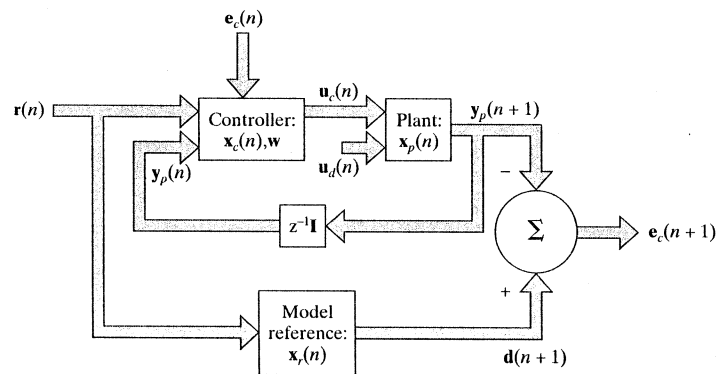


FIGURE 15.19 Model reference adaptive control using direct control.

The vector-valued function $f_2(\cdot, \cdot)$ defines the input-output behavior of the reference model.

Let the *output error* (i.e., the error between the plant and model reference outputs) be denoted by

$$e_c(n+1) = d(n+1) - y_p(n+1)$$

The design goal is to adjust the parameter vector w of the controller such that the Euclidean norm of the output error $e_c(n)$ is minimized over time n .

The method of control used in the MRAC system of Fig. 15.19 is said to be *direct* in the sense that no effort is made to identify the plant parameters, but the parameters of the controller are directly adjusted to improve system performance. Unfortunately, at present, precise methods for adjusting the parameters of the controller based on the output error are not available (Narendra and Parthasarathy, 1990). This is because the unknown plant lies between the controller and the output error. To overcome this difficulty, we may resort to the use of *indirect control*, as shown in Fig. 15.20. In this latter method, a two-step procedure is used to train the controller:

1. A model of the plant P , denoted by $\hat{P}$, is obtained to derive estimates of the differential relationships of the plant output with respect to plant input, prior plant outputs, and prior internal states of the plant. The procedure described in the previous section is used to train a neural network to identify the plant; the model $\hat{P}$ so obtained is called an *identification model*.

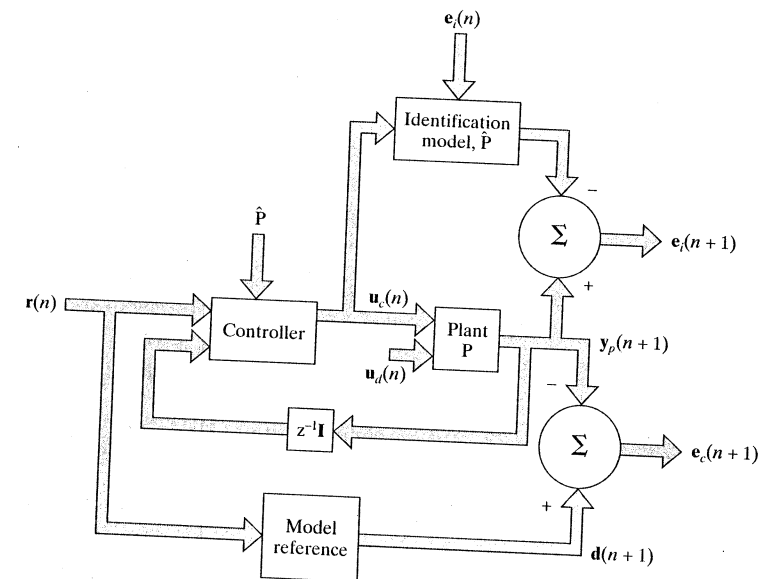


FIGURE 15.20 Model reference adaptive control using indirect control via an identification model.

2. The identification model $\hat{P}$ is used in place of the plant to derive estimates of the dynamic derivatives of the plant output with respect to the adjustable parameter vector of the controller.

In indirect control, the *externally recurrent network* is composed of the controller and an input/output representation of the plant via the identification model $\hat{P}$.

The application of a recurrent network to the controller design in the general structure of Fig. 15.20 has been demonstrated in a series of example control problems ranging from the well-known cart-pole and bioreactor benchmark problems to an automotive subsystem, namely engine idle-speed control (Puskorius and Feldkamp, 1994, Puskorius et al., 1996). The recurrent network used in those studies was a recurrent multilayer perceptron similar to that described in Section 15.2. The training of the network was performed using the DEKF algorithm described in Section 15.11. Note, however, that for the engine idle-speed control a *linear* dynamical system was chosen for the identification model since the imposed controls (over appropriately chosen ranges) appear to influence engine speed monotonically.

15 SUMMARY AND DISCUSSION

In this chapter we discuss recurrent networks that involve the use of *global feedback* applied to a static (memoryless) multilayer perceptron. The application of feedback enables neural networks to acquire state representations, making them suitable devices for diverse applications in signal processing and control. Four main network architectures belonging to the class of recurrent networks with global feedback are identified:

- Nonlinear autoregressive with exogenous inputs (NARX) networks using feedback from the output layer to the input layer.
- Fully connected recurrent networks with feedback from the hidden layer to the input layer.
- Recurrent multilayer perceptron with more than one hidden layer, using feedback from the output of each computation layer to its own input.
- Second-order recurrent networks using second-order neurons.

In all of these recurrent networks, the feedback is applied via tapped-delay-line memories.

The first three recurrent networks permit the use of a *state-space framework* for studying their dynamic behavior. This approach, rooted in modern control theory, provides a powerful method for studying the nonlinear dynamics of recurrent networks.

We describe three basic learning algorithms for the training of recurrent networks: back-propagation through time (BPTT), real-time recurrent learning (RTRL), and decoupled extended Kalman filtering (DEKF). The BPTT and RTRL algorithms are gradient based, whereas the DEKF algorithm uses higher-order information more efficiently. It is therefore able to converge much faster than the BPTT and RTRL algorithms, but at the expense of a corresponding increase in computational complexity. Indeed, the DEKF algorithm may be viewed as an enabling technology, which makes it possible to solve difficult signal processing and control problems.

In theory, a recurrent network with global feedback (e.g., recurrent multilayer perceptron trained with the DEKF algorithm) can learn the underlying dynamics of a *nonstationary* environment and do so by storing the knowledge gained from the training sample in a *fixed* set of weights. Most importantly, the network can *track* the statistical variations of the environment provided that two conditions are satisfied.

- The recurrent network does not suffer from underfitting or overfitting.
- The training sample is representative of the nonstationary behavior of the environment.

Throughout this chapter we emphasize the use of recurrent networks for temporal processing. Recurrent networks may also be used to process sequentially ordered data that do not have a straightforward temporal interpretation (e.g., chemical structures represented as trees). In Sperduti and Starita (1997) it is shown that recurrent networks can represent and classify structured patterns that are represented as directed, labeled, acyclic graphs. The main idea behind the approach described therein is the “generalized recursive neuron,” which is a structural generalization of a recurrent neuron (i.e., neuron with local feedback). By using such a model, supervised learning algorithms such as back-propagation through time and real-time recurrent learning can be extended to deal with structured patterns.

NOTES AND REFERENCES

1. For other recurrent network architectures, see Jordan (1986), Back and Tsoi (1991), Frasconi et al. (1992), and Robinson and Fallside (1991).
2. The NARX model encompasses an important class of discrete-time nonlinear systems (Leontaritis and Billings, 1985). In the context of neural networks it is discussed in Chen et al. (1990), Narendra and Parthasarathy (1990), Lin et al. (1996), and Sieglemann et al., (1997).
It has been demonstrated that the NARX model is well suited for modeling nonlinear systems such as heat exchangers (Chen et al., 1990), waste water treatment plants (Su and McAvoy, 1991; Su et al., 1992), catalytic reforming systems in a petroleum refinery (Su et al., 1992), nonlinear oscillations associated with multilegged locomotion in biological systems (Venkataraman, 1994), and grammatical inference (Giles and Horne, 1994).
The NARX model is also referred to as the nonlinear autoregressive-moving average (NARMA) model, with “moving average” referring to the inputs.
3. The recurrent multilayer perceptron in Fig. 15.4 is a generalization of the recurrent network described in Jordan (1986).
4. Omlin and Giles (1996) show that, using second-order recurrent networks, any known finite-state automata can be mapped into such a network, and the correct classification of temporal sequences of finite length is guaranteed.
5. For a rigorous treatment of controllability and observability, see Zadeh and Desoer (1963), Kailath (1980), Sontag (1990), and Lewis and Syrmos (1995).
6. The first work on neural networks and automata (actually sequential machines-automata implementations), also referenced as the first paper on finite-state automata, artificial intelligence, and recurrent neural networks, was the classic paper by McCulloch and Pitts (1943). The recurrent network (with instantaneous feedback) in the second part of this paper was interpreted as a finite-state automaton in Kleene (1956). Kleene’s paper appeared in the book “Automata Studies” edited by Shannon and McCarthy (authors in

this amazing book include Moore, Minsky, von Neumann, Uttley, McCarthy, and Shannon among others). Sometimes Kleene's paper is cited as the first article on finite-state machines (Perrin, 1990). Minsky (1967) discusses automata and neural networks in his book entitled "Computation: Finite and Infinite Machines."

All of the early work on automata and neural networks was concerned with synthesis, that is, how automata are built or designed into neural networks. Because most automata (when implemented as sequential machines) require feedback, the neural networks were necessarily recurrent ones. Note that the early work (with the exception of Minsky) did not make a clear distinction between automata (directed, labeled, acyclic graphs) and sequential machines (logic and feedback delays), and was mostly concerned with finite-state automata. There was little interest (with the exception of Minsky) in moving up the automata hierarchy to push down automata and Turing machines.

After the dark ages of neural networks, research on automata and neural networks started again in the 1980s. This work could be broadly classified into three areas: (1) learning automata, (2) automata synthesis, extraction, and refinement of knowledge, and (3) representation. The first mention of automata and neural networks was in Jordan (1986).

7. A single-layer recurrent network using McCulloch-Pitts neurons cannot simulate any finite-state machine (Goudreau et al., 1994) but Elman's simple recurrent network can (Kremer, 1995). Recurrent networks with only local feedback cannot represent all finite-state machines (Frasconi and Gori, 1996; Giles et al., 1995; Kremer, 1996).
8. The idea behind back-propagation through time is that for every recurrent network it is possible to construct a feedforward network with identical behavior over a particular time interval (Minsky and Papert, 1969). Back-propagation through time was first described in the Ph.D. thesis of Werbos (1974); see also Werbos (1990). The algorithm was rediscovered independently by Rumelhart et al. (1986b). A variant of the back-propagation through time algorithm is described in Williams and Peng (1990). For a review of the algorithm and related issues, see Williams and Zipser (1995).
9. The real-time recurrent learning algorithm was described in the neural network literature for the first time by Williams and Zipser (1989). Its origin may be traced to an earlier paper by McBride and Narendra (1965) on system identification for tuning the parameters of an arbitrary dynamical system.

The derivation given in Williams and Zipser is for a single layer of fully recurrent neurons. It has been extended to more general architectures; see, for example, Kechriotis et al. (1994); Puskorius and Feldkamp (1994).

10. Kalman filter theory owes its origin to the classic paper by Rudolf E. Kalman (1960). It has established itself as an essential part of signal processing and control with numerous applications in highly diverse fields. For a detailed treatment of the standard Kalman filter, its variants, and its extended form dealing with nonlinear dynamical systems, see Grewal and Andrews (1993) and Haykin (1996). The book by Grewal and Andrews is devoted entirely to the theory and practice of Kalman filtering. The book by Haykin discusses Kalman filter theory from the perspective of adaptive filtering. Two other important books on the subject are Jazwinski (1970) and Maybeck (1979, 1982).
11. For a detailed treatment of the square root Kalman filter and efficient methods for its implementation, see Haykin (1996).
12. Singhal and Wu (1989) were perhaps the first to demonstrate the improved mapping performance of a supervised neural network using the extended Kalman filter. Unfortunately, the training algorithm described therein is limited by its computational complexity. To overcome this limitation, Kollias and Anastassiou (1989) and Shah and Palmieri (1990) tried to simplify the application of extended Kalman filtering by parti-

tioning the global problem into a number of subproblems, each of which addresses a single neuron. However, the treatment of each neuron as an identification problem does not rigorously adhere to Kalman filter theory. Moreover, such an approach may lead to unstable behavior during training, and may result in solutions that are inferior to those obtained by other methods (Puskorius and Feldkamp, 1991).

13. Other methods for dealing with the vanishing gradients problem involve bypassing some of the nonlinearities in the recurrent network so as to provide improved learning of long-term dependencies. Examples of this approach include the following:
 - Use of long time delays in the network architecture (El Hihi and Bengio, 1996; Lin et al., 1996; Giles et al., 1997)
 - Hierarchically structuring of the network in multiple levels associated with different time scales (El Hihi and Bengio, 1996)
 - Using gating units to circumvent some of the nonlinearities (Hochreiter and Schmidhuber, 1997)
14. System identification has an extensive literature. For a treatment of the subject in book form, see Ljung (1987), and Ljung and Glad (1994). For an overview of the subject with an emphasis on neural networks, see Sjöberg et al. (1995), and Narendra (1995). The first detailed study of system identification using neural networks appeared in Narendra and Parthasarathy (1990).
15. For detailed treatment of model reference adaptive control, see the book by Landau (1979).

PROBLEMS

State-space model

- 15.1 Formulate the state-space equations for Elman's simple recurrent network shown in Fig. 15.3.
- 15.2 Show that the recurrent multilayer perceptron of Fig. 15.4 can be represented by the state-space model:

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

$$\mathbf{y}(n) = \mathbf{g}(\mathbf{x}(n), \mathbf{u}(n))$$

where $\mathbf{u}(n)$ denotes the input, $\mathbf{y}(n)$ denotes the output, $\mathbf{x}(n)$ denotes the state, and $\mathbf{f}(\cdot, \cdot)$ and $\mathbf{g}(\cdot, \cdot)$ denote vector-valued nonlinear functions.

- 15.3 Is it possible for a dynamic system to be controllable and unobservable, and vice versa? Justify your answers.
- 15.4 Referring to the local controllability problem discussed in Section 15.3, show that
 - (a) the state $\mathbf{x}(n+q)$ is a nested nonlinear function of its past value $\mathbf{x}(n)$ and the input vector $\mathbf{u}_q(n)$ of Eq. (15.24), and
 - (b) the Jacobian of $\mathbf{x}(n+q)$ with respect to $\mathbf{u}_q(n)$, evaluated at the origin, is equal to the controllability matrix $\mathbf{M}_c$ of Eq. (15.23).
- 15.5 Referring to the local observability problem discussed in Section 15.3, show that the Jacobian of the observation vector $\mathbf{y}_q(n)$ defined in Eq. (15.30) with respect to the state $\mathbf{x}(n)$, evaluated at the origin, is equal to the observability matrix $\mathbf{M}_o$ of Eq. (15.28).
- 15.6 The process equation of a nonlinear dynamical system is described by

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

where $\mathbf{u}(n)$ is the input vector at time n and $\mathbf{x}(n)$ is the corresponding state of the system. The input $\mathbf{u}(n)$ appears in the process equation in a nonadditive manner. In this problem

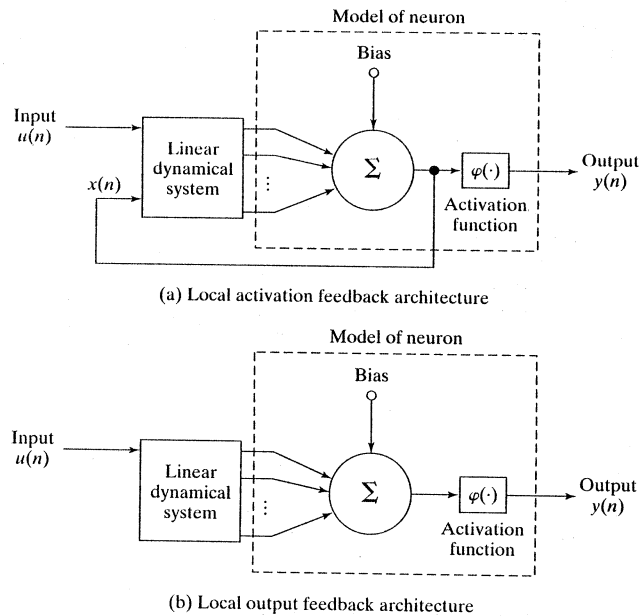


FIGURE P15.7

we wish to reformulate the process equation so that the input $u(n)$ appears additively. This is done by writing

$$\mathbf{x}'(n+1) = \mathbf{f}_{\text{new}}(\mathbf{x}'(n)) + \mathbf{u}'(n)$$

Formulate definitions for the vectors $\mathbf{x}'(n)$ and $\mathbf{u}'(n)$, and the function $\mathbf{f}_{\text{new}}(\cdot)$.

- 15.7 Figure P15.7 presents two examples of recurrent network architectures using local feedback at the neuronal level. The architectures shown in parts a and b of the figure are called *local activation feedback* and *local output feedback*, respectively (Tsoi and Back, 1994). Formulate state-space models for these two recurrent network architectures. Comment on their controllability and observability.

Nonlinear autoregressive with exogenous inputs (NARX) model

- 15.8 Referring to the NARX model discussed in Section 15.4, show that the use of Eqs. (15.16) and (15.17) leads to the following expression for the output $y(n+q)$ of the NARX model in terms of the state $\mathbf{x}(n)$ and input vector $\mathbf{u}_q(n)$:

$$y(n+q) = \Phi(\mathbf{x}(n), \mathbf{u}_q(n))$$

where $\Phi: \mathbb{R}^{2q} \rightarrow \mathbb{R}$, and $\mathbf{u}_q$ is defined in accordance with Eq. (15.29).

- 15.9 (a) The derivation of the NARX model in Section 15.4 is presented for a single input, single output system. Discuss how the theory described therein can be extended for a multiple input, multiple output system.

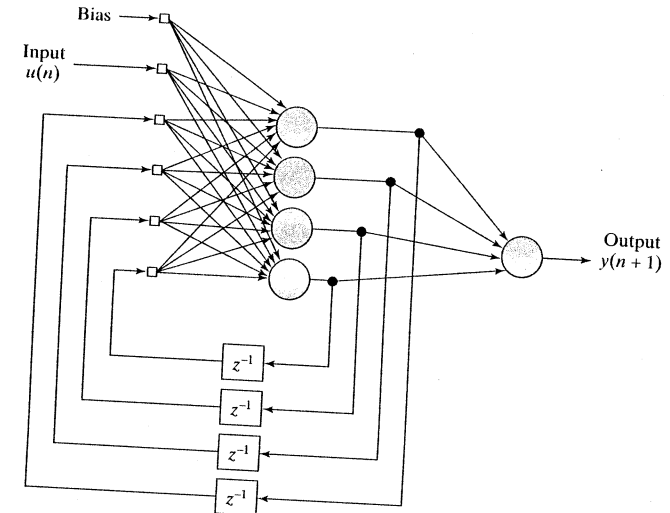


FIGURE P15.10

- (b) Construct the NARX equivalent to the two input, single output state-space model in Fig. 15.6.

- 15.10 Construct the NARX equivalent for the fully recurrent network shown in Fig. P15.10.
- 15.11 In Section 15.4 we showed that any state-space model can be represented by a NARX model. What about the other way around? Can any NARX model be represented by a state-space model of the form described in Section 15.3? Justify your answer.

Back-propagation through time

- 15.12 Unfold the temporal behavior of the state-space model shown in Fig. 15.3.
- 15.13 The truncated BPTT(h) algorithm may be viewed as an approximation to the epochwise BPTT algorithm. The approximation can be improved by incorporating aspects of epochwise BPTT into the truncated BPTT(h) algorithm. Specifically, we may let the network through h' additional steps before performing the next BPTT computation, where $h' < h$. The important feature of the hybrid form of back-propagation through time is that the next backward pass is not performed until time step $n+h'$. In the intervening time, past values of the network input, network state, and desired responses are stored in a buffer, but no processing is performed on them (Williams and Peng, 1990). Formulate the local gradient for neuron j in this hybrid algorithm.

Real-time recurrent learning algorithm

- 15.14 The dynamics of a *teacher forced recurrent network* during training are as described in Section 15.8, except for this change:

$$\xi_i(n) = \begin{cases} u_i(n) & \text{if } i \in \mathcal{A} \\ d_i(n) & \text{if } i \in \mathcal{C} \\ y_i(n) & \text{if } i \in \mathcal{B} - \mathcal{C} \end{cases}$$

where $\mathcal{A}$ denotes the set of indices i for which ξ_i is an external input, $\mathcal{B}$ denotes the set of indices j for which ξ_j is the output of a neuron, and $\mathcal{C}$ denotes the set of output neurons that are visible.

- (a) Show that for this scheme, the partial derivative $\partial y_j(n+1)/\partial w_{ki}(n)$ is given by (Williams and Zipser, 1989)

$$\frac{\partial y_j(n+1)}{\partial w_{ki}(n)} = \varphi'(v_j(n)) \left[\sum_{i \in \mathcal{B} - \mathcal{C}} w_{ji}(n) \frac{\partial y_i(n)}{\partial w_{ki}(n)} + \delta_{kj} \xi_i(n) \right]$$

- (b) Derive the training algorithm for a teacher forced recurrent network.

Decoupled Extended Kalman Filtering (DEKF) algorithm

15.15 Describe how the DEKF algorithm can be used to train the simple recurrent network shown in Fig. 15.3. You may also invoke the use of the BPTT algorithm for this training.

15.16 In its usual form, DEKF training is carried out with weight updates, instance by instance. By contrast, in standard back-propagation, simple gradient updates are performed, enabling us to choose to apply the updates immediately or else accumulate the updates for some time and then apply them as a single composite update. Although such an accumulation could be attempted in the DEKF algorithm, doing so would cause inconsistency between the weight vector and the error covariance matrix, which is updated each time a recursion is performed to generate a weight update. Thus the use of DEKF training appears to preclude batch updating. However, it is possible to use *multistream DEKF training*, which allows for multiple training sequences and yet remains consistent with Kalman filter theory, as described in Feldkamp et al. (1997) and Feldkamp and Puskorius (1998).

- (a) Consider the training problem with N_{in} inputs, N_{out} outputs, and a fixed training sample of N examples. From the training sample, form $M \leq N$ data streams which feed M networks constrained to have identical weights. At each training cycle, one pattern from each stream is presented to its respective network and the N_{out} network outputs for each stream are computed. A single weight update is then computed and applied identically to each stream's network. Derive the multistream form of the DEKF algorithm.
- (b) For example, consider the standard XOR problem with four training patterns. Assume that we have a feedforward network that is augmented with a delay-line memory connected to the output layer. We thus effectively have four network outputs: the actual network output fed into the delay-line memory, and three delayed versions of it with each one of them constituting a new network output. Now apply each of the four training patterns in some order to this network structure, but do not perform any weight updates. After the presentation of the fourth training pattern, we have four network outputs that represent the processing of the four training patterns performed through a network with identical weights. If we consider performing a single weight vector update with DEKF on the basis of these four training patterns and four network outputs, we have a four-stream problem. Check this example out.

Second-order recurrent networks

15.17 In this problem we explore the construction of the *parity finite-state automaton* using a second order recurrent network. This automaton recognizes an odd number of 1's in an arbitrary length sequential string of 0's and 1's.

Figure P15.17 shows a two-state automaton. States are represented by circles, and transitions by arrows. S means we start in that state, state A in the case shown here. The thick circle means that whenever we are in that state, shown as state B in the figure, we

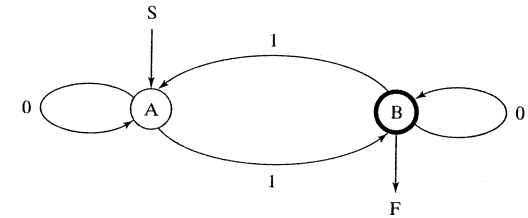


FIGURE P15.17

accept the string. The automaton starts seeing strings in state A and loops back to state A if we see a 0 and loops to state B if we see a 1. Similarly, when in state B , it loops to state B if we see a 0 and loops back to state A if we see a 1. In this way, the automaton is always in state A if we have seen an even (including zero) number of 1's and in state B if we have seen an odd number of 1's.

More formally, we define the states as $Q = \{A, B\}$, $S = A$ as the start state, the input alphabet as $\Sigma = \{0, 1\}$, the accepting state as $F = B$, and the state transition function as:

$$\delta(A, 0) = A$$

$$\delta(A, 1) = B$$

$$\delta(B, 0) = B$$

$$\delta(B, 1) = A$$

These are the equations needed for the application of Eq. (15.9) pertaining to the second-order recurrent network. For more details on finite-state automata, see Hopcroft (1979).

Encode the above transition rules into the second-order recurrent network.

15.18 In Section 15.8 we derive the real-time recurrent learning (RTRL) algorithm for a fully connected recurrent network using first-order neurons. In Section 15.2 we describe a recurrent network using second-order neurons.

Extend the theory described in Section 15.8 by deriving the RTRL algorithm for the training of a second-order recurrent network.