

# **HANDBOOK OF INTELLIGENT CONTROL**

## **NEURAL, FUZZY, AND ADAPTIVE APPROACHES**

Edited by  
David A. White  
Donald A. Sofge

1992



VAN NOSTRAND REINHOLD

\_\_\_\_\_  
New York

# FOREWORD<sup>1</sup>

This book is an outgrowth of discussions that got started in at least three workshops sponsored by the National Science Foundation (NSF):

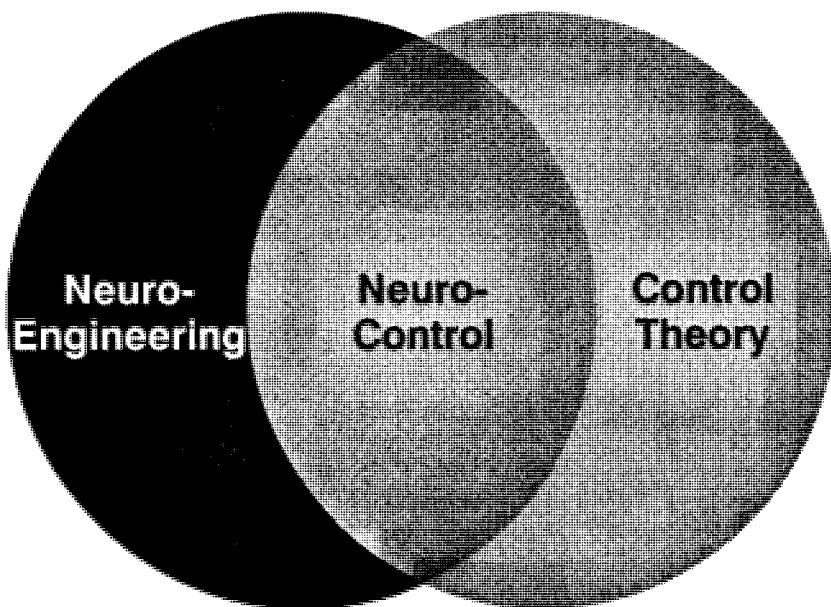
- A workshop on neurocontrol and aerospace applications held in October 1990, under joint sponsorship from McDonnell Douglas and the NSF programs in Dynamic Systems and Control and Neuroengineering
- A workshop on intelligent control held in October 1990, under joint sponsorship from NSF and the Electric Power Research Institute, to scope out plans for a major new joint initiative in intelligent control involving a number of NSF programs
- A workshop on neural networks in chemical processing, held at NSF in January–February 1991, sponsored by the NSF program in Chemical Reaction Processes

The goal of this book is to provide an authoritative source for two kinds of information: (1) fundamental new designs, at the cutting edge of true intelligent control, as well as opportunities for future research to improve on these designs; (2) important real-world applications, including test problems that constitute a challenge to the *entire* control community. Included in this book are a series of realistic test problems, worked out through lengthy discussions between NASA, NeuroDyne, NSF, McDonnell Douglas, and Honeywell, which are more than just benchmarks for evaluating intelligent control designs. Anyone who contributes to solving these problems may well be playing a crucial role in making possible the future development of hypersonic vehicles and subsequently the economic human settlement of outer space. This book also emphasizes chemical process applications (capable of improving the environment as well as increasing profits), the manufacturing of high-quality composite parts, and robotics.

The term “intelligent control” has been used in a variety of ways, some very thoughtful, and some based on crude attempts to market aging software. To us, “intelligent control” should involve both *intelligence* and *control theory*. It should be based on a serious attempt to understand and replicate the phenomena that we have always called “intelligence”—i.e., the generalized, flexible, and adaptive kind of capability that we see in the human brain. Furthermore, it should be firmly rooted in control theory to the fullest extent possible; admittedly, our development of new designs must often be highly intuitive in the early stages, but, once these designs are specified, we should at least do our best to understand them and evaluate them in terms of the deepest possible mathematical theory. This book tries to maintain that approach.

---

<sup>1</sup> The views expressed here are those of the authors and do not represent official NSF views. The figures have been used before in public talks by the first author.



**Figure F.1** Neurocontrol as a subset.

Traditionally, intelligent control has embraced classical control theory, neural networks, fuzzy logic, classical AI, and a wide variety of search techniques (such as genetic algorithms and others). This book draws on all five areas, but more emphasis has been placed on the first three.

Figure F.1 illustrates our view of the relation between control theory and neural networks. Neurocontrol, in our view, is a *subset* both of neural network research *and* of control theory. None of the basic design principles used in neurocontrol is totally unique to neural network design; they can all be understood—and improved—more effectively by viewing them as a subset and extension of well-known underlying principles from control theory. By the same token, the new designs developed in the neurocontrol context can be applied just as well to classical nonlinear control. The bulk of the papers on neurocontrol in this book discuss neurocontrol in the context of control theory; also, they try to provide designs and theory of importance to those control theorists who have no interest in neural networks *as such*. The discussion of biology may be limited here, but we believe that these kinds of designs—designs that draw on the power of control theory—are likely to be more powerful than some of the simpler, more naive connectionist models of the past; therefore, we suspect that they will prove to be more relevant to actual biological systems, which are also very powerful controllers. These biological links have been discussed extensively in other sources, which are cited in this book.

Those chapters that focus on adaptive control and neurocontrol implicitly assume the following definition: Intelligent control is the use of *general-purpose* control systems, which *learn* over time how to achieve goals (or optimize) in *complex, noisy, nonlinear* environments whose dynamics must

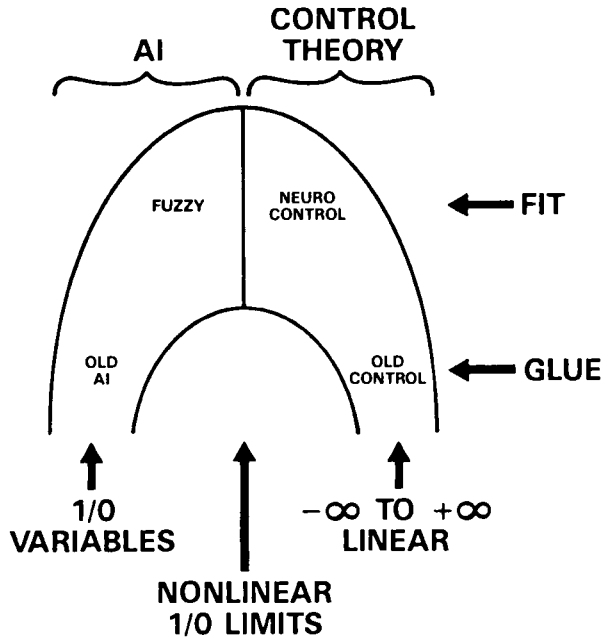


Figure F.2 Aspects of intelligent control.

ultimately be learned in real time. This kind of control cannot be achieved by simple, incremental improvements over existing approaches. It is hoped that this book provides a blueprint that will make it possible to achieve such capabilities.

Figure F.2 illustrates more generally our view of the relations between control theory, neurocontrol, fuzzy logic, and AI. Just as neurocontrol is an innovative subset of control theory, so too is fuzzy logic an innovative subset of AI. (Some other parts of AI belong in the upper middle part of Figure F.2 as well, but they have not yet achieved the same degree of prominence in engineering applications.) Fuzzy logic helps solve the problem of human-machine communications (in querying experts) and formal symbolic reasoning (to a far less extent in current engineering applications).

In the past, when control engineers mainly emphasized the linear case and when AI was primarily Boolean, so-called intelligent control was mainly a matter of cutting and pasting: AI systems and control theory systems communicated with each other, in relatively ad hoc and distant ways, but the fit was not very good. Now, however, fuzzy logic and neurocontrol both build *nonlinear* systems, based on *continuous* variables bounded at 0 and 1 (or  $\pm 1$ ). From the controller equations alone, it becomes more and more difficult to tell which system is a neural system and which is a fuzzy system; the distinction begins to become meaningless in terms of the mathematics. This moves us towards a new era, where control theory and AI will become far more compatible with each other. This allows arrangements like what is shown in Figure F.3, where neurocontrol and fuzzy logic can be used as *two complementary sets of tools for use on one common controller*.

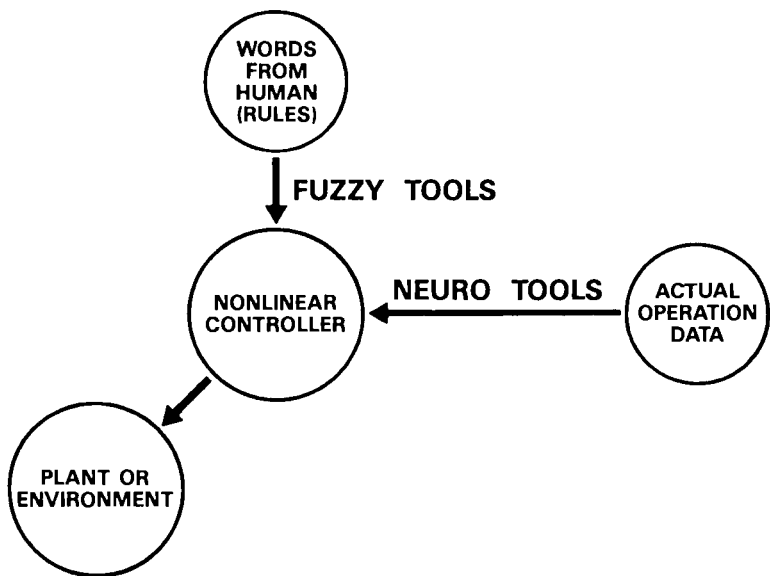


Figure F.3 A way to combine fuzzy and neural tools.

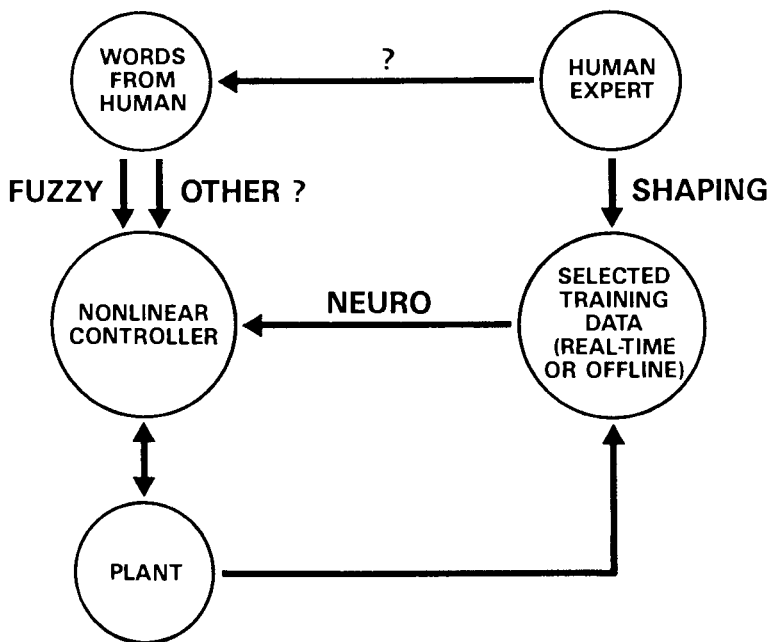


Figure F.4 A way to combine fuzzy and neural tools.

In practice, there are many ways to combine fuzzy logic and other forms of AI with neurocontrol and other forms of control theory. For example, see Figure F.4.

This book will try to provide the basic tools and examples to make possible a wide variety of combinations and applications, and to stimulate more productive future research.

*Paul J. Werbos*

NSF Program Director for Neuroengineering and  
Co-director for Emerging Technologies Initiation

*Elbert Marsh*

NSF Deputy A.D. for Engineering and  
Former Program Director for Dynamic Systems and Control

*Kishan Baheti*

NSF Program Director for Engineering Systems and  
Lead Program Director for the Intelligent Control Initiative

*Maria Burka*

NSF Program Director for Chemical Reaction Processes

*Howard Moraff*

NSF Program Director for Robotics and Machine Intelligence

# NEUROCONTROL AND SUPERVISED LEARNING: AN OVERVIEW AND EVALUATION

*Paul J. Werbos*

*NSF Program Director for Neuroengineering  
Co-director for Emerging Technologies Initiation<sup>1</sup>*

## 3.1. INTRODUCTION AND SUMMARY

### 3.1.1. General Comments

This chapter will present detailed procedures for using adaptive networks to solve certain common problems in adaptive control and system identification. The bulk of the chapter will give examples using *artificial neural networks* (ANNs), but the mathematics are general. In place of ANNs, one can use *any* network built up from differentiable functions or from the solution of systems of nonlinear equations. (For example, section 3.2 will show how one can apply these procedures to econometric models, to fuzzy logic structures, etc.) Thus, the methods to be developed here are really part of learning control or adaptive control in the broadest sense. For those with no background in neural networks, it may be helpful to note that everything in this chapter may be viewed as approximations or extensions of dynamic programming or to other well-known techniques, developed to make it possible to handle large, noisy, nonlinear problems in real time, and to exploit the computational power of special-purpose neural chips.

In the past, many applications of ANNs to control were limited to the use of static or feedforward networks, adapted in relatively simple ways. Many applications of that sort encountered problems such as large error rates or limited application or slow adaptation. This chapter will try to summarize

---

<sup>1</sup> The views expressed here are those of the author, and not the official views of NSF.

these problems and their solutions. Many of these problems can be solved by using *recurrent* networks, which are essentially just networks in which the output of a neuron (or the value of an intermediate variable) may depend on its own value in the recent past. Section 3.2 will discuss recurrent networks in more detail. For adaptation in real time, the key is often to use *adaptive critic* architectures, as in the successful applications by White and Sofge in Chapters 8 and 9. To make adaptive critics learn quickly on large problems, the key is to use more advanced adaptive critics, which White and Sofge have used successfully, but have not been exploited to the utmost as yet.

### 3.1.2. Neurocontrol: Five Basic Design Approaches

Neurocontrol is defined as the use of well-specified neural networks—artificial or natural—to emit actual control signals. Since 1988, hundreds of papers have been published on neurocontrol, but virtually all of them are still based on five basic design approaches:

- Supervised control, where neural nets are trained on a database that contains the “correct” control signals to use in sample situations
- Direct inverse control, where neural nets directly learn the mapping from desired trajectories (e.g., of a robot arm) to the control signals which yield these trajectories (e.g., joint angles) [1,2]
- Neural adaptive control, where neural nets are used instead of linear mappings in standard adaptive control (see Chapter 5)
- The backpropagation of utility, which maximizes some measure of utility or performance over time, but cannot efficiently account for noise and cannot provide real-time learning for very large problems [3,4,5]
- Adaptive critic methods, which may be defined as methods that approximate dynamic programming (i.e., approximate optimal control over time in noisy, nonlinear environments)

Outside of this framework, to a minor extent, are the feedback error learning scheme by Kawato [6] which makes use of a *prior* feedback controller in direct inverse control, a scheme by McAvoy (see Chapter 10) which is a variant of the backpropagation of utility to accommodate constraints, and “classical conditioning” schemes rooted in Pavlovian psychology, which have not yet demonstrated an application to optimal control or other well-defined control problems in engineering [7]. A few other authors have treated the problem of optimal control over time as if it were a simple function minimization problem, and have used random search techniques such as simulated annealing or genetic algorithms.

Naturally there are many ways to combine the five basic designs in complex real-world applications. For example, there are many complex problems where it is difficult to find a good controller by adaptation alone, starting from random weights. In such problems, it is crucial to use a strategy called “shaping.” In shaping, one first adapts a simpler controller to a simplified version of the problem, perhaps by using a simpler neurocontrol approach or even by talking to an expert; then, one uses the weights of the resulting controller as the *initial values* of the weights of a controller to solve the more complex problem. This approach can, of course, be repeated many times if necessary. One can also build systems that phase in gradually from a simpler approach to a more complex approach. Hierarchical combinations of different designs are also discussed in other chapters of this book.



All of the five basic methods have valid uses and applications, reviewed in more detail elsewhere [8,9]. They also have important limitations.

Supervised control and direct inverse control have been reinvented many times, because they both can be performed by directly using *supervised learning*, a key concept in the neural network field. All forms of neurocontrol use supervised learning networks as a basic building block or module in building more complex systems. *Conceptually*, we can discuss neurocontrol without specifying *which* of the many forms of supervised learning network we use to fill in these blocks in our flow charts; as a practical matter, however, this choice has many implications and merits some discussion. The concept of propagating derivatives *through* a neural network will also be important to later parts of this book. After the discussion of supervised learning, this chapter will review current problems with each of the five design approaches mentioned above.

## 3.2. SUPERVISED LEARNING: DEFINITIONS, NOTATION, AND CURRENT ISSUES

### 3.2.1. Supervised Learning and Multilayer Perceptrons

Many popularized accounts of ANNs describe supervised learning as if this were the only task that ANNs can perform.

Supervised learning is the task of learning the mapping from a vector of inputs,  $X(t)$ , to a vector of targets or desired outputs,  $Y(t)$ . "Learning" means adjusting a vector of weights or parameters,  $W$ , in the ANN. "Batch learning" refers to any method for adapting the weights by analyzing a fixed database of  $X(t)$  and  $Y(t)$  for times  $t$  from  $t = 1$  through  $T$ . "Real-time learning" refers to a method for adapting the weights *at* time  $t$ , "on the fly," as the network is running an actual application; it inputs  $X(t)$  and then  $Y(t)$  at each time  $t$ , and adapts the weights to account for that *one* observation.

Engineers sometimes ask at this point: "But what *is* an ANN?" Actually, there are many forms of ANN, each representing possible functional forms for the relation that maps from  $X$  to  $Y$ . In theory, we could try to do supervised learning for *any* function or mapping  $f$  such that:

$$\hat{Y} = f(X, W). \quad (1)$$

Some researchers claim that one functional form is "better" than another in some universal sense for all problems; however, this is about as silly as those economists who claim that *all* equations must have logarithms in them to be relevant to the real world.

Most often, the functional mapping from  $X$  to  $Y$  is modeled by the following equations:

$$x_i = X_i \quad 1 \leq i \leq m \quad (2)$$

$$net_i = \sum_{j=1}^{i-1} W_{ij} x_j \quad m < i \leq N + n \quad (3)$$

$$x_i = 1/(1 + \exp(-net_i)) \quad m < i \leq N + n \quad (4)$$

$$\hat{y}_i = x_{i+N} \quad 1 \leq i \leq n, \quad (5)$$

where  $m$  is the number of inputs,  $n$  is the number of outputs, and  $N$  is a measure of how big you choose to make the network.  $N-n$  is the number of “hidden neurons,” the number of intermediate calculations that are not output from the network.

Networks of this kind are normally called “multilayer perceptrons” (MLP). Because the summation in equation 3 only goes up to  $i-1$ , there is no “recurrence” here, no cycle in which the output of a “neuron” ( $x_i$ ) depends on its own value; therefore, these equations are *one example* of a “feedforward” network. Some authors would call this a “backpropagation network,” but this confuses the meaning of the word “backpropagation,” and is frowned on by many researchers. In this scheme, it is usually crucial to delete (or zero out) connections  $W_{ij}$ , which are not necessary. For example, many researchers use a three-layered network, in which all weights  $W_{ij}$  are zeroed out, except for weights going from the input layer to the hidden layer and weights going from the hidden layer to the output layer.

### 3.2.2. Advantages and Disadvantages of Alternative Functional Forms

Conventional wisdom states that MLPs have the best ability to approximate arbitrary nonlinear functions but that backpropagation—the method used to adapt them—is much slower to converge than are the methods used to adapt alternative networks. In actuality, the slow learning speed has a lot to do with the *functional form itself* rather than the backpropagation algorithm, which can be used on a wide variety of functional forms.

MLPs have several advantages: (1) given enough neurons, they can approximate any well-behaved bounded function to an arbitrarily high degree of accuracy, and can even approximate the less well-behaved functions needed in direct inverse control [10]; (2) VLSI chips exist to implement MLPs (and several other neural net designs) with a very high computational throughput, equivalent to Crays on a chip. This second property is crucial to many applications. MLPs are also “feedforward” networks, which means that we can calculate their outputs by proceeding from neuron to neuron (or from layer to layer) without having to solve systems of nonlinear equations.

Other popular forms of feedforward neural networks include CMAC (see Chapters 7 and 9), radial basis functions (see Chapter 5), and vector quantization methods [11]. These alternatives all have a key advantage over MLPs that I would call *exclusivity*: The network calculates its output in such a way that only a *small number of localized weights* have a strong effect on the output at any one time. For different regions of the input-space (the space of  $X$ ), different sets of weights are invoked. This has tremendous implications for real-time learning.

In real-time learning, there are always dangers of thrashing about as the system moves from one large region of state-space to another, unlearning what it learned in previous regions [12].

For example, in learning a quadratic function, a system may move occasionally from one region to another; if a *linear* approximation fits well enough within any one region, the system may virtually ignore the quadratic terms while simply changing its estimates of the linear terms as it moves from region to region. It may take many cycles through the entire space before the network begins to exploit the quadratic terms. Networks with *exclusivity* tend to avoid this problem, because they adapt *different weights* in different regions; however, they also tend to contain more weights and to be poor at

*generalizing* across different regions. See Chapter 10 for a full discussion of reducing the number of weights and its importance to generalization.

Backpropagation, and least squares methods in general, has shown much better real-time learning when applied to networks with high exclusivity. For example, DeClaris has reported faster learning (in feedforward pattern recognition) using backpropagation with networks very similar to radial basis functions [13]. Jacobs et al. [14], reported fast learning, using a *set* of MLPs, coordinated by a maximum likelihood classifier that assumes that each input vector  $X$  belongs to one and only one MLP. Competitive networks [15]—a form of simultaneous-recurrent network—also have a high degree of exclusivity. From a formal mathematical point of view, the usual CMAC learning rule is equivalent to backpropagation (least squares) *when applied* to the CMAC functional form.

To *combine* fast learning and good generalization in real-time supervised learning, I once proposed an approach called “syncretism” [16], which would combine: (1) a high-exclusivity network, adapted by ordinary real-time supervised learning, which would serve in effect as a long-term *memory*; (2) a more parsimonious network, trained in real time but *also* trained (in occasional periods of batch-like “deep sleep”) to match the memory network in the regions of state-space well-represented in memory. Syncretism is probably crucial to the capabilities of systems like the human brain, but no one has done the research needed to implement it effectively in artificial systems. Ideas about learning and generalization *from examples*, at progressively higher levels of abstraction, as studied in human psychology and classical AI, may be worth reconsidering in this context. Syncretism can work only in applications—like supervised learning—where high-exclusivity networks make sense.

Unfortunately, the most powerful controllers cannot have a high degree of exclusivity in all of their components. For example, there is a need for *hidden units* that serve as feature detectors (for use across the entire state-space), as filters, as sensor fusion layers, or as dynamic memories. There is no way to adapt such units as quickly as we adapt the output nodes. (At times, one can learn features simply by clustering one’s inputs; however, this tends to break down in systems that truly have a high dimensionality—such as 10—in the set of realizable states, and there is no guarantee that it will generate optimal features.) Human beings can learn to recognize individual objects very quickly (in one trial), but even they require *years* of experience during infancy to build up the basic feature recognizers that underlie this ability.

This suggests that the optimal neural network, even for real-time learning, will typically consist of multiple layers, in which the upper layer can learn quickly and the lower layer(s) must learn more slowly (or even off-line). For example, one may build a two-stage network, in which the first stage is an MLP—a design well-suited for generalized feature extraction; the output of the MLP could then be used as the input to a differentiable CMAC, such as the one developed by Sofge and White [17].

Finally, one can achieve even more computational power—at the cost of even more difficult adaptation—by adding *recurrence* to one’s networks. There are actually two forms of recurrent networks: (1) time-lagged recurrent networks (TLRNs), in which neurons may input the outputs from *any* other neurons (or from themselves) from a previous clock cycle; (2) simultaneous-recurrent networks, which still perform a static mapping from  $X$  to  $Y$ , but are based on a kind of iterative relaxation process. TLRNs are essential when building neural networks for system identification, or even for the recognition of dynamic patterns. This chapter will show how TLRNs for system identification are essential to *all* forms of neurocontrol, but it will not show how to adapt them; those details are provided in Chapter 10, which will also discuss how to *combine* time-lagged recurrence

and simultaneous recurrence in a single network. On the other hand, since simultaneous-recurrent networks are essentially just another way to define a static mapping, they will be discussed further at the end of this section.

### 3.2.3. Methods to Adapt Feedforward Networks

The most popular procedure for adapting ANNs is called *basic backpropagation*. Figure 3.1 illustrates an application of this method to supervised control, where the inputs ( $X$ ) are usually sensor readings and the targets ( $u^*$ ) are *desired* control responses to those sensor readings. (In this application, the targets  $Y$  happen to be control signals  $u^*$ .)

At each time  $t$ , basic backpropagation involves the following steps:

1. Use equations 1 through 4 to calculate  $u(t)$ .
2. Calculate the error:

$$E(t) = 1/2 \sum_{i=1}^n (\hat{Y}_i(t) - Y_i(t))^2. \quad (6)$$

3. Calculate the derivatives of error with respect to all of the weights,  $W_{ij}$ . These derivatives may be denoted as  $F_{W_{ij}}$ . (This kind of notation is especially convenient when working with multiple networks or networks containing many kinds of functions.)
4. Adjust the weights by steepest descent:

$$W_{ij}(T+1) = W_{ij}(t) - LR * F_{W_{ij}}(t), \quad (7)$$

where  $LR$  is a learning rate picked arbitrarily ahead of time, *or adapt the weights* by some other gradient-based technique [18,19].

Backpropagation, in its most general form, is simply the method used for calculating all the derivatives efficiently, in one quick backwards sweep through the network, symbolized by the dashed line in Figure 3.1. Backpropagation can be applied to *any* network of differentiable functions. (See Chapter 10.) Backpropagation *in this sense* has many uses that go far beyond the uses of basic backpropagation.

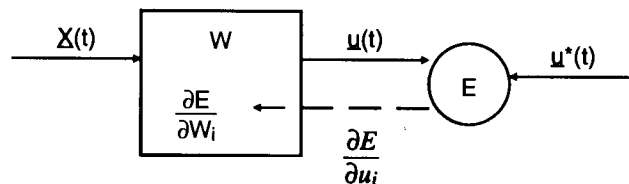


Figure 3.1 Basic backpropagation in supervised control.

Conventional wisdom states that basic backpropagation—or gradient-based learning in general—is a very slow but accurate method. The accuracy comes from the minimization of square error, which is consistent in spirit with classical statistics. Because of the slow speed, backpropagation is sometimes described as irrelevant to real-time applications.

In actuality, steepest descent can converge reasonably rapidly, *if* one takes care to *scale* the various weights in an appropriate way and *if* one uses a reasonable method to adapt the learning rate [20]. For example, I have used backpropagation in a real production system that required convergence to four decimal places in less than forty observations [5], and White and Sofge have reported the use of backpropagation in an Action network in real-time adaptation [17]. Formal numerical analysis has developed many gradient-based methods that do far better than steepest descent in *batch* learning, but for real-time learning at  $O(N)$  cost per time period there is still very little theory available.

Based on these considerations, one should at least replace equation 7 by:

$$W_{ij}(t+1) = W_{ij}(t) - LR(t) * S_{ij} * F_{-}W_{ij}(t), \quad (8)$$

where the  $S_{ij}$  are scaling factors. We do not yet know how best to adjust the scaling factors automatically in real time, but it is usually possible to build an off-line version of one's control problem in order to experiment with the scaling factors and the structure of the network. If the scaling factors are set to speed up learning in the experiments, they will usually lead to much faster learning in real time as well. To adapt the learning rate, one can use the procedure [5]:

$$LR(t+1) = a*LR(t) + b*LR(t)*\left(\frac{F_{-}W(t+1) \cdot F_{-}W(t)}{|F_{-}W(t)|^2}\right), \quad (9)$$

where  $a$  and  $b$  are constants and the dot in the numerator refers to a vector dot product.

Theoretically,  $a$  and  $b$  should be 1, but in batch learning experiments I have found that an “ $a$ ” of 0.9 and a “ $b$ ” of .2 or so usually work better. With sparse networks, equation 9 is often enough by itself to yield adequate performance, even without scaling factors.

In some applications (especially in small tests of pattern classification!), this procedure will be disturbed by an occasional “cliff” in the error surface, especially in the early stages of adaptation. To minimize this effect, it is sometimes useful to create a filtered average of  $|F_{-}W|^2$  and scale down any gradient whose size exceeds this average by a factor of 3 or more; in the extreme, one may even normalize the gradient to unit length. Likewise, in real-time learning applications, the past gradient in equation 9,  $F_{-}W(t)$ , should probably be replaced by some kind of filtered value. Once again, there are more rigorous and powerful methods available for batch learning [18,19], but they have yet to be extended to real-time learning.

A crucial advantage of equation 9 is that it can be applied *separately* to *separate* groups of weights. For example, one can use equation 9 to *independently* adapt a different learning rate for different layers of a network. This is crucial to many practical applications, where hidden and/or recurrent layers require slow adaptation.

To complete this section, I must show how to calculate the derivatives,  $F_{-}W_{ij}(t)$ , in equation 8. For any differentiable feedforward network, one may perform these calculations as follows:

$$F_{-}^{\wedge} Y_i(t) = \frac{\partial E(t)}{\partial \hat{Y}_i(t)} = \hat{Y}_i(t) - Y_i(t) \quad (10)$$

$$\{F_{-}W_{ij}(t)\} = F_{-}f_w(X, W, F_{-}^{\wedge}Y), \quad (11)$$

where  $F_{-}f_w$  is a function calculated by the *dual subroutine* of the neural network  $f$ .

Appendix B of Chapter 10 explains how to program the dual subroutine for *any* differentiable feedforward network. In the special case of MLPs, the equations for the dual subroutine are:

$$F_{-}x_i(t) = F_{-}\hat{Y}_{i-N}(t) + \sum_{j=i+1}^{N+m} W_{ji} * F_{-}net_j(t) \quad i = N+m, \dots, 1 \quad (12)$$

$$F_{-}net_i(t) = s'(net_i(t)) * F_{-}x_i(t) \quad i = N+m, \dots, m+1 \quad (13)$$

$$F_{-}W_{ij}(t) = F_{-}net_i(t) * x_j(t), \quad \text{all } W_{ij} \quad (14)$$

where:

$$s'(net_i(t)) = x_i(t) * (1 - x_i(t)), \quad (15)$$

where the first term on the right-hand side of equation 12 is taken as zero for  $i \leq N$ , and where  $F_{-}net_j$  is taken as zero for  $j \leq m$ . Notice how this dual subroutine would generate *two* useful output arrays,  $F_{-}W$  and  $F_{-}X$  (which is the same as  $F_{-}x_i$  for  $i \leq m$ ); the main inputs to the dual subroutine are simply the three arguments of the dual function used in equation 11. (The terms  $x_i$  and  $net_i$ , needed in equations 13 and 14, can either be recalculated from  $X$  and  $W$  or retrieved from a previous run of the original network  $f$  based on the same  $X$  and  $W$ .)

Likewise, for a two-stage network where  $\hat{Y} = f(z)$  and  $z = g(X, W)$ , we can calculate the required derivatives simply by calling the dual subroutines for the two subnetworks  $f$  and  $g$ :

$$F_{-}z = F_{-}f_z(z, F_{-}\hat{Y}) \quad (16)$$

$$\{F_{-}W_{ij}\} = F_{-}g_w(X, W, F_{-}z).$$

Few researchers have actually used dual subroutines in adapting simple MLPs for pattern recognition. It is easier just to code equations 10, 12, 13, 14, and 15 into one big subroutine. However, in complex control applications, dual subroutines are essential in order to maintain understandable modular designs. Intuitively, dual subroutines may be thought of as subroutines that propagate derivatives (*any* derivatives) back from the outputs of a network to its inputs. They backpropagate to *all* of the inputs in one sweep, but to represent them as functions we need to use subscripts (like the  $W$  in  $F_{-}f_w$ ) to identify which input is being used.

### 3.2.4 Adapting Simultaneous-Recurrent or Simultaneous-Equation Systems

Simultaneous-recurrent networks are extremely popular in some parts of the neural network community. Following the classical work of Grossberg and Hopfield, networks of this kind are usually represented as differential equation models. For our purposes, it is easier and more powerful to represent these kinds of networks as *iterative* or *relaxation* systems. More precisely, a simultaneous-recurrent network will be defined as a mapping:

$$\hat{Y} = F(X, W), \quad (17)$$

which is computed by iterating over the equation:

$$y^{(n+1)} = f(y^{(n)}, X, W) \quad (18)$$

where  $f$  is some sort of feedforward network or system, and  $\hat{Y}$  is defined as the equilibrium value of  $y(y^{(\infty)})$ . For some functions  $f$ , there will exist no stable equilibrium; therefore, there is a huge literature on how to guarantee stability by special choices of  $f$ . Many econometric models, fuzzy logic inference systems, and so on calculate their outputs by solving systems of nonlinear equations that can be represented in the form of equation 18; in fact, equation 18 can represent almost any iterative update procedure for any vector  $y$ .

To use equation 11 in adapting this system (or to use the control methods described below), all we need to do is program the dual subroutine  $F\_F$  in an efficient manner. One way to calculate the required derivatives—proposed by Rumelhart, Hinton, and Williams in 1986 [21]—required us to store *every* iterate,  $y^{(n)}$ , for  $n = 1$  to  $n = \infty$ . In the early 1980s, I developed and applied a more efficient method, but my explanation of the general algorithm was very complex [22]. Using the concept of *dual subroutine*, however, a much simpler explanation is possible.

Our goal is to build a dual subroutine,  $F\_F$ , which inputs  $F\_Y$  and outputs  $F\_X$  and/or  $F\_W$ . To do this, we must first program the dual subroutine,  $F\_f$ , for the feedforward network  $f$  that we were iterating over in equation 18. Then we may simply iterate over the equation:

$$F\_y^{(n+1)} = F\_Y + F\_f(\hat{Y}, X, W, F\_y^{(n)}). \quad (19)$$

In cases where the original iteration over equation 18 converged, we are guaranteed that this dual iteration will converge as well, usually faster [22]. (Note that  $F\_y^{(n)}$  is only our  $n$ th *estimate* of  $F\_y$ ; it has *no relation* to our earlier iterate  $y^{(n)}$ .) After convergence has been obtained, we may go on to calculate:

$$F\_X = F\_f_x(\hat{Y}, X, W, F\_y^{(\infty)}). \quad (20)$$

$$\{F\_W_{ij}\} = F\_f_w(\hat{Y}, X, W, F\_y^{(\infty)}). \quad (21)$$

The computer program for  $F\_F$  would therefore consist of a DO loop over  $n$ , implementing equation 19, followed by two simple subroutine calls.

As an example, consider the case where:

$$f(X, y, W_x, W_y) = W_x X + W_y y. \quad (22)$$

It is easy to verify that the procedure above yields the correct derivatives in this linear example (which can also be used in testing computer code):

$$y^{(\infty)} = W_x X + W_y y^{(\infty)} \quad (23)$$

$$\hat{Y} = (I - W_y)^{-1} W_x X \quad (24)$$

$$F_{-y}(\hat{Y}, X, W_x, W_y, F_{-y}^{(\infty)}) = F_{-y} \hat{Y} + W_y^T F_{-y}^{(\infty)} = (I - W_y^T)^{-1} F_{-y} \hat{Y} \quad (25)$$

$$F_{-x}(\hat{Y}, X, W_x, W_y, F_{-y} \hat{Y}) = W_x^T F_{-y}^{(\infty)} = W_x^T (I - W_y^T)^{-1} F_{-y} \hat{Y} \quad (26)$$

The validity of the procedure in the general case follows directly from equations 22 through 26, applied to the corresponding Jacobians.

In practical applications of simultaneous-recurrent networks, it is desirable to train the network so as to minimize the number of iterations required for adequate convergence and to improve the chances of stability. Jacob Barhen, of the Jet Propulsion Laboratory, has given presentations on "terminal teacher forcing," which can help do this. The details of Barhen's method appear complicated, but one might achieve a very similar effect by simply adding an additional term representing lack of convergence (such as  $(y^{(N+1)} - y^{(N)})^2$ , where  $N$  is the last iteration) to the error function to be minimized and is an "arbitrary" weighting factor. The parameter may be thought of as a "tension" parameter; high tension forces quicker decisions or tighter convergence, while lower tension—by allowing more time—may sometimes yield a better final result. Intuitively, we would expect that the optimal level of tension varies over time, and that the management of tension should be a key consideration in shaping the learning experience of such networks. However, these are new concepts, and I am not aware of any literature providing further details.

The practical value (and potential hazards) of using simultaneous-recurrent networks is described in Chapters 10 and 13.

### 3.3. FIVE BASIC DESIGN APPROACHES IN NEUROCONTROL

#### 3.3.1 Supervised Control: Common Problems

Supervised control seems very simple to implement, as described above. In practice, it presents a number of challenges.

The first challenge is to build up the database of sensor inputs and desired actions. If we already *know* what actions to take in a wide variety of situations, doesn't that mean that we *already* have an adequate controller? If so, what good is the neural net equivalent? In actuality, there are many situations where it is useful to *copy* a known controller with a neural network. For example, one might copy the skills of a human expert, so as to "clone" him or speed him up [8]. Conventional expert systems copy what a person *says* to do, but supervised control copies what a person *actually does* (as recorded in the database). Likewise, supervised control can be used to copy a computer-based controller that runs very well on a Cray at slow speed but requires a smaller, faster copy for real-world



applications. To understand a supervised control system, it is essential to find out where the database came from, because the neural net is only trying to copy the person or system who generated the database.

The second challenge lies in handling dynamics. There are many problems where simple static networks or models can give an adequate description of the human expert; however, there are many problems where they are inadequate. For example, Tom McAvoy has pointed out that good human operators of chemical plants are precisely those operators who do a good job of responding to dynamic trends. *Human controllers—like the automatic controllers I will discuss—cannot adapt to changing parameters like changing temperatures and pressures—without an ability to respond to dynamics, as well.* In other words, the advantages of time-lagged recurrent networks in automatic control apply as well to human controllers and to ANN clones of those human controllers.

For optimal performance, therefore, supervised control should *not* be treated as a simple exercise in static mapping. It should be treated as an exercise in *system identification*, an exercise in *adapting a dynamic model* of the human expert. Chapter 10 describes at length the techniques available for system identification by neural network.

### 3.3.2. Common Problems With Direct Inverse Control

Figure 3.2 illustrates a typical sort of problem in direct inverse control. The challenge is to build a neural network that will input a *desired location*— $X(t)$ —specified by a higher-level controller or a human, and output the control signals,  $u(t)$ , which will move the robot arm to the desired location.

In Figure 3.2, the desired location is in two-dimensional space, and the control signals are simply the joint angles  $\theta_1$  and  $\theta_2$ . If we ignore dynamic effects, then it is reasonable to assume that  $x_1$  and  $x_2$  are a function,  $f$ , of  $\theta_1$  and  $\theta_2$ , as shown in the figure. Furthermore, if  $f$  happens to be invertible (i.e., if we can solve for  $\theta_1$  and  $\theta_2$  uniquely when given values of  $x_1$  and  $x_2$ ), then we can use the inverse mapping  $f^{-1}$  to tell us how to choose  $\theta_1$  and  $\theta_2$  so as to reach any desired point  $(x_1^*, x_2^*)$ .

Most people using direct inverse control begin by building a database of *actual*  $X(t)$  and  $u(t)$  simply by flailing the arm about; they train a neural net to learn the inverse mapping from  $X(t)$  to  $u(t)$ . In other words, they use supervised learning, with the actual  $X(t)$  used as the inputs and the actual  $u(t)$  used as the targets. Most users find ways to avoid the obvious problems in random flailing about, but there are three problems that are prevalent in this kind of work:

- Positioning errors tend to be about four to five percent of the workspace, which is inadequate for industrial robotic applications. (For example, see [23].)

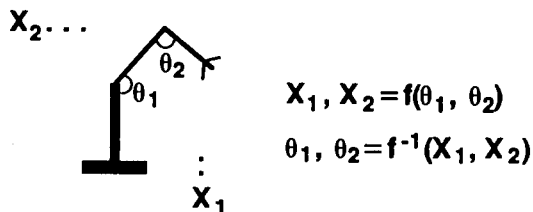


Figure 3.2 Direct inverse control.

- Even without adding extra actuators, the mapping  $f$  in Figure 3.2 is typically not invertible, which—as Kawato [6] and Jordan [24] have shown—can lead to serious errors.
- There is no ability to exploit extra actuators to achieve maximally smooth, graceful, or energy-saving motion.

Two approaches have been found to solve the first of these problems: (1) use of a highly accurate supervised learning system (which can interfere with real-time learning) and a robot arm with minimal dynamic effects [25]; (2) use of a cruder, faster, supervised learning system, and the inclusion of  $X(t-1)$  in addition to  $X(t)$  as an input to the neural network. Even if the neural net itself has errors on the order of five to ten percent, the second approach could allow a *moving* robot arm to *reduce* tracking error at time  $t$  to five or ten percent of *what they were at time  $t-1$* . In other words, the dynamic approach puts a feedback loop around the problem; it captures the capabilities of classical controllers based on feedback, but it also can allow for nonlinearity. Using an approach of this general sort, Tom Miller has reported tracking errors for a real Puma robot that were a tiny fraction of a percent; see Chapter 7 for details.

Nevertheless, Miller's scheme has no provision for minimizing energy use or the like. That would require an optimal control method such as backpropagating utility (which Kawato and Jordan have applied to these robot motion problems) or adaptive critics.

Miller has shown that his robot is relatively adaptive to changes in parameters, parameters such as the mass of an unstable cart which his robot arm pushes around a figure-8 track. After a mass change, the robot arm returns to accurate tracking after only three passes around the track. Still, it may be possible to do far better than this with recurrent networks. Suppose that Miller's network had hidden nodes or neurons that implicitly measured the mass, in effect, by trying to track the dynamic properties of the cart. Suppose further that this network was trained over a *series* of mass changes, to allow the correct adaptation of those hidden nodes. In that case, the robot should be able to adapt near-instantaneously to new changes in mass. In summary, one could improve on Miller's approach by treating the direct inverse problem as a problem in system identification, forecasting  $Q(t)$  over time with a recurrent neural network.

### 3.3.3. Adaptive Control and Neural Networks

The standard textbook on adaptive control in the United States is *Stable Adaptive Systems* by Narendra and Annaswamy [26], which grew out of pioneering work by Narendra in the 1970s. The techniques given in that book are mainly the model reference adaptive control (MRAC) techniques, but the book also suggests that these techniques are very similar for our purposes to the self-tuning regulator designs pioneered by Åström in Sweden.

Classical MRAC techniques solve a problem very similar to that of direct inverse control. The user or higher-order controller is asked to supply a *Reference Model* that outputs a desired trajectory. The control system is asked to send control signals to a robot arm or to some other kind of "plant," so as to make the plant follow the desired trajectory. It is usually assumed that the plant and the controller are both *linear*, but that the parameters of the plant are not known ahead of time. Narendra has shown how to *adapt* the parameters of the control system to solve this problem.

There are two practical problems that this work has solved, which are important to neurocontrol as well: (1) the problem of whole-system stability; (2) the problem of adapting to changes in parameters of the system to be controlled.

The problem of whole-system stability is crucial to many real-world applications. For example, the people who own a billion-dollar chemical plant want strong assurances that a control system *cannot* destroy this plant through instability. If ANNs provide a one percent savings in cost, but cannot provide proofs of stability, there may be difficulties in getting acceptance from them. In aerospace applications, there are similar verification issues. Many neural network researchers have experience in using Lyapunov methods to prove the stability of neural networks *as such* [27]; however, the issue here is the stability of the *whole system* made up of the unknown plant, the controller, and the adaptation procedure all put together. Proofs of stability have been very hard to find in the nonlinear case, in classical control theory. The desire to go beyond the linear case is one of the key motivations of Narendra's work with ANNs, discussed in Chapter 5.

The need to adapt to unexpected changes and changes in parameters has been even more crucial to the interest of industry both in adaptive control and in neurocontrol. Because ANNs can adapt their parameters in real time, it is hoped that they will be able to adapt in real time to unexpected events. In actuality, there are at least three types of unexpected events one could try to adapt to:

1. Measurable noise events, such as measurable but unpredictable turbulence as a well-instrumented airplane climbs at high speed
2. Fundamental but normal parameter changes, such as unmeasured drifts in mechanical elasticities or temperature that normally occur over time in operating a system
3. Radical structural changes, such as the loss of a wing of an airplane, which were totally unanticipated in designing and training the controller

The ideal optimal controller should do as well as possible in coping with all three kinds of events; however, it is often good enough in practice just to handle type 1 or type 2 events, and there are severe limits to what is possible for *any* kind of control design in handling type 3 problems in the real world.

Real-time learning can help with all three types of events, but it is really crucial only for type 3. For type 1, it may be of minimal value. Type 3 events may also require special handling, using fast associative memories (e.g., your past experience flashes by in an instant as your plane dives) and idiosyncratic systems that take over in extreme circumstances. One way to improve the handling of type 3 events is to try to anticipate what *kinds* of things might go wrong, so as to make them more like type 2 events. The best way to handle type 1 events is to use a control system which is explicitly designed to account for the presence of noise, such as an adaptive critic system.

Real-time learning can help with type 2 events, but it may not be the fastest and best way to go. For example, the previous discussion of direct inverse control gave an example of adapting to changing masses. If we train a network over data in which masses and other parameters are changing, certain types of network can learn to *recognize* familiar kinds of parameter change. If mass changes are familiar from past experience, it should be possible to adapt to them much more quickly than it would if they were a type 3 event.

This kind of adaptiveness is possible *only* for networks that contain neurons or units that are capable of responding to changes in the dynamics of the plant. (How to *adapt* such units is another issue, described in Chapter 10.) For example, one such intermediate unit might be needed to estimate the temperature of the plant. To do this, that unit must somehow be able to integrate information over long periods of time, because plant dynamics usually cannot be measured accurately based only on data from  $t$  and  $t-1$ . Some researchers might simply add a whole lot of tapped delay lines, inputs going

back from  $t$  and  $t-1$  through to  $t-1000$ ; however, this results in a very nonparsimonious network, which causes severe problems in learning, as described above. It is better to use structures like those used in Kalman filtering, in which the state of the intermediate variables at time  $t$  depends on observed inputs at  $t$  and  $t-1$  and *on their own state* at time  $t-1$ . This, in turn, implies that the system must be a *time-lagged recurrent network*.

The ideal approach is to *combine* the best techniques for handling all three types of events—adaptive critic controllers, with recurrent networks and real-time learning. In many applications, however, real-time learning or recurrent networks may be good enough by themselves. For example, one may design a controller based on recurrent networks, and adapt it *offline* using the backpropagation of utility; the resulting controller may still be “adaptive” in a significant way. Likewise, one may build a predictive model of a plant and use backpropagation through time to adapt it *off-line*; then, one may freeze the weights coming into the memory neurons and use them as a kind of fixed preprocessor inside of a system that learns in real time.

Narendra has developed three designs so far that extend MRAC principles to neural networks. The first is essentially just direct inverse control, adapted very slightly to allow the desired trajectory to come from a reference model. The second—which he called his most powerful and general in 1990—is very similar to the strategy used by Jordan [24]. In this approach, the problem of following a trajectory is converted into a problem in optimal control, simply by trying to minimize the gap between the desired trajectory and the actual trajectory. This is equivalent to *maximizing* the following utility function (which is always negative):

$$U = -1/2 \sum_{ij} (X_i(t) - X_i^*(t))^2, \quad (27)$$

where  $X(t)$  is the actual position at time  $t$  and  $X^*$  is the desired or reference position. Narendra then uses the backpropagation of utility to *solve* this optimization problem. Jordan has pointed out that one can take this approach further by adding (negative) terms to the utility function which represent jerkiness of the motion and so on. Furthermore, it is possible to use adaptive critics instead to solve this optimization problem, as first proposed in [9].

Many authors refer to equation 27 as an *error* measure, but it is easier to keep the different modules of a control system straight if you think of error as something you do in forecasting or pattern classification, etc.; utility refers to desirable or undesirable *physical* conditions in the external world [28,29]. Keeping these concepts distinct becomes important when we move on to the backpropagation of utility and adaptive critics; in those designs, we usually propagate derivatives of utility *through* a Model network for various applications, but the weights in the Model network itself are adapted in response to *error*. By expressing equation 27 as a utility function, we can also appreciate more easily the value of adding extra terms to that utility function, as in the work of Kawato and Jordan. In many applications (as above), it is a little easier to think about minimizing *cost* or *disutility*, rather than maximizing utility, because of the sign reversal; however, this chapter will emphasize maximization for the sake of internal consistency.

Narendra's third design appears for the first time in this book. The third design is similar to the second design, except that radial basis functions are used in place of an MLP, and certain constraints are placed on the system; within these constraints, Narendra provides a rigorous proof of whole-sys-

tem convergence. In the future, there is also some hope of developing stability proofs for adaptive critic systems, by adapting a Critic network to serve as a kind of constructed Lyapunov function. There is also some possibility of validating neural networks for specific applications by using tests more like those now used to qualify human operators for the same tasks (subject to automatic emergency overrides).

### 3.3.4. Backpropagating Utility and Recurrent Networks

Section 3.1 mentioned two methods for maximizing utility over future time: the backpropagation of utility and adaptive critics. The backpropagation of utility is an exact and straightforward method, essentially equivalent to the calculus of variations in classical control theory [30]. The backpropagation of utility can be used on two different kinds of tasks: (1) to adapt the parameters or weights,  $W$ , of a controller or Action network  $A(x, W)$ ; (2) to adapt a *schedule* of control actions ( $u$ ) over future time. The former approach—first proposed in 1974 [31]—was used by Jordan in his robot arm controller [24] and by Widrow in his truck-backer-upper [4]. The latter approach was used by Kawato in his cascade method to control a robot arm [6] and by myself, in an official 1987 DOE model of the natural gas industry [5]; Chapter 10 gives more recent examples which, like [5], involve optimization subject to constraints. This section will emphasize the former approach.

Both versions begin by assuming the availability of an exact *model* or *emulator* of the plant to be controlled, which we may write as:

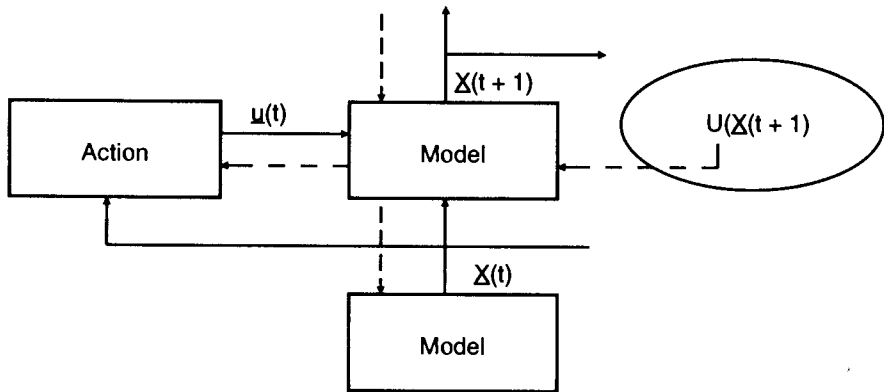
$$x(t+1) = f(x(t), u(t)). \quad (28)$$

This model may itself be an ANN, adapted to predict the behavior of the plant. In practice, we can adapt the Action network and the Model network concurrently, but this chapter will only describe how to adapt the Action network.

The backpropagation of utility proceeds as if the Model network were an exact, noise-free description of the plant. (This assumption may or may not be problematic, depending on the application and the quality of the model.) Based on this assumption, the problem of maximizing total utility over time becomes a straightforward, if complex, problem in function maximization. To maximize total utility over time, we can simply calculate the derivatives of utility with respect to all of the weights, and then use steepest ascent to adjust the weights until the derivatives are zero. This is usually done *off-line*, in batch mode.

The basic idea is shown in Figure 3.3. A simple implementation of Figure 3.3, using the concept of *dual* subroutines discussed in section 3.2 and Chapter 10, is:

1. Repeat the following five steps until convergence.
2. Initialize  $F_W$  and Total\_Utility to zero.
3. For  $t$  going from 1 (initial time) to  $T$  (final time), do:
  - a.  $u(t) = A(x(t), W)$
  - b.  $x(t+1) = f(x(t), u(t))$
  - c. Total\_Utility = Total\_Utility +  $U(x(t+1))$ .
4. Set  $F_x(T+1) = F_{U_x}(x(T+1))$  (i.e.,  $F_{x_i}(t+1) = \partial U(x(t+1)) / \partial x_i(t+1)$ ).
5. For  $t$  from  $T$  to 1, do:
  - a.  $F_u(t) = F_{f_u}(x(t), u(t), F_x(t+1))$  (i.e., backpropagate through the model).



**Figure 3.3** Backpropagating utility, using backpropagation through time (BTT).

- b.  $F_{\mathbf{x}}(t) = F_{\mathbf{U}_x}(\mathbf{x}(t)) + F_{f_x}(\mathbf{x}(t), \mathbf{u}(t), F_{\mathbf{x}}(t+1))$  (backpropagate through the model, adding grad  $\mathbf{U}$ ).
- c.  $F_{\mathbf{W}} = F_{\mathbf{W}} + F_{A_w}(\mathbf{x}(t), F_{\mathbf{u}}(t))$  (propagate derivatives through the Action network).
6. Update  $\mathbf{W} = \mathbf{W} + \text{learning\_rate} * F_{\mathbf{W}}$ .

Step 5 is a straightforward application of the chain rule for ordered derivatives, discussed in Chapter 10. All of the “ $F_{\mathbf{W}}$ ” arrays contain derivatives of total utility; they should not be confused with the very different derivatives used in adapting the Model network itself.

Strictly speaking, we should add a third term— $F_{A_x}(\mathbf{x}(t), F_{\mathbf{u}}(t))$ —to the right-hand side of step 5b. This term is important only when the Action network cannot accurately represent an optimal controller (e.g., when it is an MLP with too few weights).

Figure 3.3 represents a use of *backpropagation through time*, which is normally used in an offline mode because of the calculations backwards in time in step 5. It is possible, instead, to use a *forwards perturbation* method [36] to calculate the derivatives; however, this usually increases the computational costs by a factor of  $n$ , where  $n$  is the number of weights in the Action network. Such an increase in cost is bearable if  $n$  is on the order of ten to 100, and if the cost of calculation is not a large concern. Many practical control problems do have that characteristic; therefore, Narendra’s recent examples—which use this approach—point towards a useful form of real-time learning and control for such problems. Likewise, if the required planning horizon is only ten to 100 steps into the future, then the alternative method in Chapter 10 may be useful in real time.

There are tricks to account for noise in these methods, but their numerical efficiency in the general case is questionable.

No matter how we calculate the derivatives of utility here, we still need to use recurrent networks in order to make the scheme fully adaptive, as discussed earlier. The logical way to do this is by adapting a Model network that contains neurons able to input their own state at time  $t-1$ . The input vector to the Action network should represent the complete state vector of the plant, including both the observables  $\mathbf{X}$  and the outputs  $\mathbf{R}$  of the hidden units in the Model network. This approach is similar to the usual dual-control scheme based on Kalman filtering [30]. Once again, the challenge is to adapt recurrent networks for system identification.

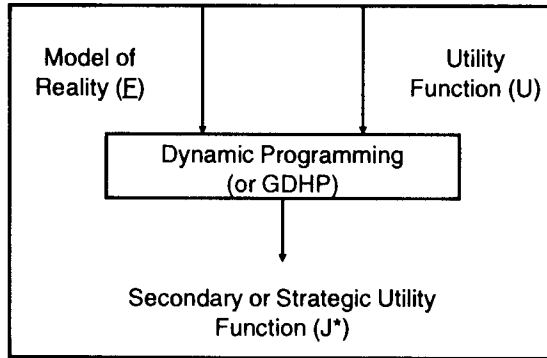


Figure 3.4 Inputs and outputs of dynamic programming.

### 3.3.5. Adaptive Critics

The adaptive critic family of designs is more complex than the other four. The simplest adaptive critic designs learn slowly on large problems but have generated many real-world success stories on difficult small problems. Complex adaptive critics may seem intimidating, at first, but they are the only design approach that shows serious promise of duplicating critical aspects of human intelligence: the ability to cope with large numbers of variables in parallel, in real-time, in a noisy nonlinear environment.

Adaptive critic designs may be defined as designs that attempt to approximate dynamic programming in the general case. Dynamic programming, in turn, is the *only* exact and efficient method for finding an optimal strategy of action over time in a noisy, nonlinear environment. The cost of running true dynamic programming is proportional (or worse) to the number of possible states in the plant or environment; that number, in turn, grows exponentially with the number of variables in the environment. Therefore, approximate methods are needed even with many small-scale problems. There are many variations of dynamic programming; Howard's textbook [32] is an excellent introduction to the iterative, stochastic versions important here.

Figure 3.4 illustrates the basic concept common to all forms of dynamic programming. The user supplies a utility function  $U$  and a *stochastic* model,  $F$ , of the plant or environment to be controlled. Dynamic programming is used to solve for another function,  $J$ , which serves as a secondary or strategic utility function. The key theorem is that any strategy of action that maximizes  $J$  in the short term will also maximize the sum of  $U$  over all future times.  $J$  is a function of  $R(t)$ , where  $R$  is a complete state description of the plant to be controlled at time  $t$ . Adaptive critic designs are defined more precisely as designs that include a Critic network—a network whose output is an approximation to the  $J$  function, or to its derivatives, or to something very closely related to these two. The Action network in an adaptive critic system is adapted so as to maximize  $J$  in the near-term future. To maximize future utility *subject to constraints*, you can simply train the Action network to obey those constraints when maximizing  $J$ ; the validity of dynamic programming itself is not affected by such constraints.

To actually build an adaptive critic control system, you must decide on two things: (1) exactly what the critic network is supposed to approximate, and how you will adapt it to make it do so; (2)

how you will adapt the Action network in response to information coming out of the Critic network. This section will summarize the range of choices on both of these issues, starting with the choice of Critics. Chapters 6 and 13 will give more complete information on how to adapt a Critic network, but the information in this section is reasonably complete for Action networks.

As of 1979, there were three Critic designs that had been proposed based on dynamic programming:

1. Heuristic dynamic programming (HDP), which adapts a Critic network whose output is an approximation of  $J(\mathbf{R}(t))$  [16,33]. The temporal difference method of Barto, Sutton, and Anderson [34] turns out to be a special case of HDP (see Chapter 13).
2. Dual heuristic programming (DHP), which adapts a Critic network whose outputs represent the *derivatives* of  $J(\mathbf{R}(t))$  [16,35]. The derivative of  $J(\mathbf{R}(t))$  with respect to  $R_i(t)$  is usually written as  $\lambda_i(\mathbf{R}(t))$ .
3. Globalized DHP (GDHP), which adapts a Critic network whose output is an approximation of  $J(\mathbf{R}(t))$ , but adapts it so as to minimize errors in the implied *derivatives* of  $J$  (as well as  $J$  itself, with some weighting) [35,36,37]. GDHP tries to combine the best of HDP and DHP.

HDP tends to break down, through very slow learning, as the size of a problem grows bigger; however, DHP is more difficult to implement. A major goal of this chapter and of Chapter 13 is to stimulate research on DHP by explaining the method in more detail.

The three methods listed above all yield *action-independent* critics. In all cases, the Critic network inputs the vector  $\mathbf{R}(t)$ , but not the vector of actions,  $\mathbf{u}(t)$ , for the sake of consistency with dynamic programming (see Figures 3.5 through 3.7). In 1989, Lukes, Thompson, and myself [38,39] and Watkins [40] independently arrived at methods to adapt *action-dependent* critics, shown in Figure 3.8. To do this, we went back to the most relevant form of the Bellman equation [32], the equation that underlies dynamic programming:

$$J(\mathbf{R}(t)) = \max_{\mathbf{u}(t)} (U(\mathbf{R}(t), \mathbf{u}(t)) + \langle J(\mathbf{R}(t+1)) \rangle / (1+r) - U_0), \quad (29)$$

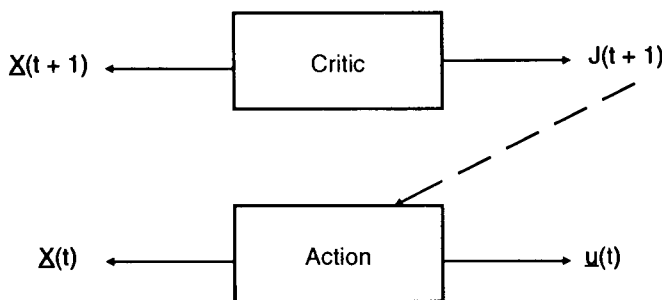
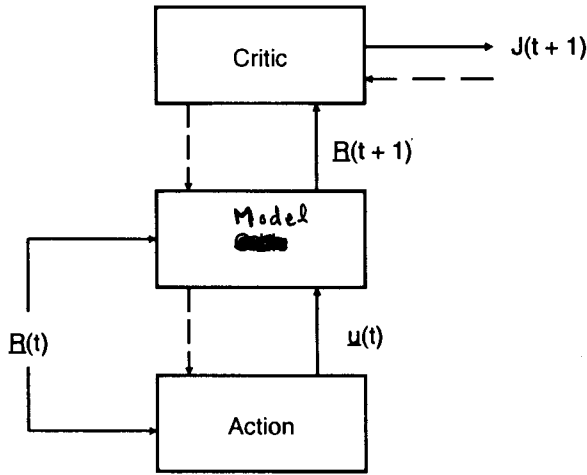


Figure 3.5 The two-network design of Barto, Sutton, and Anderson.





**Figure 3.6** The Backpropagated Adaptive Critic (with J-style Critic).

where  $r$  and  $U_0$  are constants that are needed only in infinite-time-horizon problems (and then only sometimes), and where the angle brackets refer to expectation value. We then defined a new quantity:

$$J'(R(t), u(t)) = U(R(t), u(t)) + \langle J(R(t+1)) \rangle / (1+r) \quad (30)$$

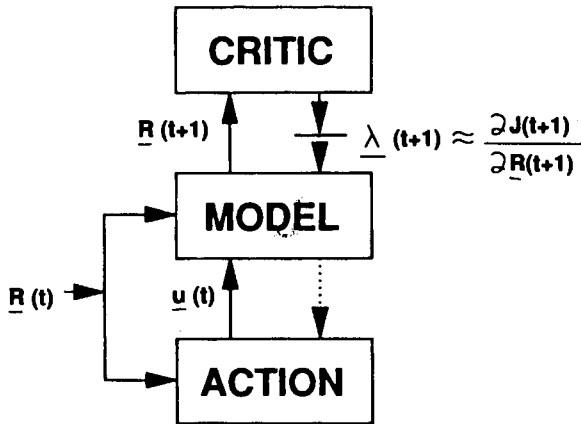
By substituting between these two equations, we may derive a recurrence rule for  $J'$ :

$$J'(R(t), u(t)) = U(R(t), u(t)) + \max_{u(t+1)} \langle J'(R(t+1), u(t+1)) \rangle / (1+r) - U_0. \quad (31)$$

Both Watkins and our group developed a way to adapt a Critic network that inputs  $R(t)$  and  $u(t)$ , and outputs an estimate of  $J'(R(t), u(t))$ . We called this an action-dependent adaptive critic (ADAC), and Watkins called it Q-learning. Actually, it would be more precise to call the adaptation procedure action-dependent HDP (ADHDP), because the procedure is almost identical to HDP. Chapter 13 will also describe an action-dependent version of DHP (ADDHP).

In addition, tricks have been proposed to make these methods work better in practice [33,34,35,37]. For example [35,37], if  $U$  can be written as a sum of known components,  $U_i$ , one can then estimate a separate  $J_i$  function for each of the  $U_i$ . In theory, the sum of these  $J_i$  functions is just another way to express the overall  $J$  function, but if each of the  $J_i$  components requires fewer inputs than the total  $J$  does, one can expect greater accuracy and faster learning by adapting the components separately. (In statistical terms, fewer independent variables usually imply smaller standard errors [41].) White and Sofge have found this trick to be very useful in their work. This trick is essentially just a way station on the way from HDP to DHP, which effectively breaks out even more components.

Whichever method we use to adapt the Critic network, we still need to find a way to adapt the Action network in response to the Critic. Until 1990, virtually all of the applications of adaptive critics



**Figure 3.7** The Backpropagated Adaptive Critic (with  $\lambda$ -style Critic and  $U$  independent of  $u(t)$ ).

were based on the simple scheme used by Barto, Sutton, and Anderson, illustrated in Figure 3.5. In this scheme, the Critic network emits a single scalar, an estimate of  $J(R(t))$ . This scalar is then used as a kind of gross “reward” or “punishment” signal to the Action network. More precisely, the weights in the Action network are all adapted by using an algorithm called A-rp [34], which implements the notion of reward or punishment.

This scheme has worked very well with small problems. With moderate-sized problems, however, it can lead to very slow learning—a problem that can be fixed by using more advanced designs. Whenever the vector  $u(t)$  has several components, the scheme in Figure 3.5 does not tell us *which* component should be rewarded when things go well, or punished if they do not. Furthermore, even if we know that the  $u_i(t)$  component of  $u$  was the really important action, we still don’t know whether  $u_i$  should have been greater or smaller. The more action variables are, the more serious this problem becomes. Intuitively, this is like telling a student that he or she has done “well” or “badly” without grading the individual problems on a test; it makes it difficult for the student to improve performance. (In formal statistical terms [41], one would say that there is a severe problem of multicollinearity when we try to explain a single dependent variable— $J$ —on the basis of a very large number of actions or weights.) Clearly, we can do better if we can get additional information indicating *which* action to change, in *which* direction. In other words, we can do better if we somehow obtain information about *derivatives*. (Essentially the same argument explains why DHP can do better than HDP in adapting the Critic network itself.)

By 1990, two alternative methods had been used to overcome this problem in adapting an Action network. The first—the backpropagated adaptive critic (BAC)—is illustrated in Figures 3.6 and 3.7. (Figure 3.6 is for more conventional Critic networks, adapted by HDP or GDHP, while Figure 3.7 is for Critics adapted by DHP.) BAC implements the idea of trying to pick  $u(t)$  so as to maximize  $J(t+1)$ , which is what equation 29 tells us to do. The idea is to build up a model of the plant to be controlled, and use backpropagation to calculate the derivatives of  $J(t+1)$  with respect to the weights in the Action network. These weights can then be adapted by steepest ascent or by some other gradient-based method. Jameson [42] has shown that the design in Figure 3.6 does work, but realistic and large-scale

tests of performance with better Model networks are still needed. BAC assumes an action-independent critic network. BAC can be used either in an on-line mode (where the actual value of  $R(t+1)$  is input to the Critic at time  $t+1$ ) or in a “dreaming” or exploratory mode (where  $R(t)$  is chosen “at random” and  $R(t+1)$  is predicted by the model). Either way, Figure 3.6 implies derivative calculations that can be understood by referring back to our discussion of Figure 3.3:

$$\begin{aligned} F_R(t+1) &= F_{JR}(R(t+1)) \\ F_u(t) &= F_{fR}(R(t), u(t), F_R(t+1)) \\ F_W &= F_{Aw}(R(t), W, F_u(t)). \end{aligned} \quad (32)$$

(Note that equation 32 contains an additional term,  $F_{Uu}$ , not shown in Figure 3.6, which is needed only when the utility function  $U$  is a function of  $u$  as well as  $R$ ;  $F_{Uu}$  represents the output of a subroutine which calculates the derivatives of  $U$  with respect to  $u$ .)

When controlling a plant which changes fairly little from time  $t$  to time  $t+1$ , it is possible to improve robustness by changing the action  $u$  as well as the weights  $W$  in response to  $F_u$ , as in the cerebellum [44] and in the “bias” term of Chapter 8.

The second method—the ADAC approach—is shown in Figure 3.8. With an ADAC critic network, one maximizes  $J'(R(t), u(t))$  directly as a function of  $u(t)$ , as suggested by equation 18. To do this, one can use backpropagation directly from the Critic to the Action network, as shown in Figure 3.8. Jordan and Jacobs [43] and Sofge and White [17] used this approach in 1990 to adapt an Action network. Sofge and White reported great success with this approach.

Hybrids of these various approaches are also possible. For example, ADAC systems might possibly be used to account for errors in more model-dependent designs, so as to make them more resistant to model errors and to type 3 unexpected events [35, p. 82].

What is the need for recurrent networks here? It certainly is not obvious that recurrent networks are needed in Figure 3.8! In Figures 3.6 and 3.7, it is obvious that we do need a Model network with BAC. As in adaptive control and in the backpropagation of utility, we can expect to achieve more truly adaptive control if the Model network contains recurrent neurons. Such neurons can learn to represent unknown parameters, and they also can provide a kind of short-term memory, in which our predictions for  $X(t+1)$  may depend on phenomena we observed a few time cycles ago that happen to be out of current sensor range.

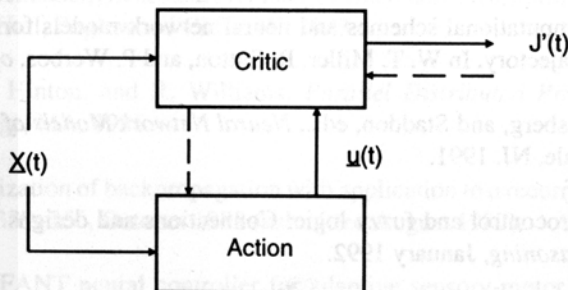


Figure 3.8 Adapting the Action network with ADAC.

Even in Figure 3.8, recall that dynamic programming requires the  $R$  vector to be a *complete state vector*. Even with ADAC, our ability to develop a high degree of adaptiveness or short-term memory requires that we construct an  $R$  vector that provides these capabilities. The best way to do this is simply to adapt a Model network and use its hidden nodes as an expanded state vector. One could do something similar by using Kalman filtering on the observed sensor data,  $X(t)$ , but the neural net offers a more general nonlinear formulation of the same ideas.

In summary, the adaptive critic family of methods is a large and complex family, ranging from simple and well-tested, but limited, methods, to more complex methods that eventually promise true brain-like intelligent performance. Early applications of the latter have been very promising, but there is an enormous need for more tests, more studies of applications demanding enough to justify the greater complexity, and more creative basic research. Better system identification, based on methods like those of Chapter 10, is crucial to the capabilities of all of these systems; indeed, it is crucial to all of the five basic approaches of neurocontrol.

### 3.4. REFERENCES

- [1] L. G. Kraft and D. Campagna, A summary comparison of CMAC neural network and traditional adaptive control systems. *Neural Networks for Control*, W. T. Miller, R. Sutton, and P. Werbos, MIT Press, Cambridge, MA, 1990.
- [2] A. Guez and J. Selinsky, A trainable neuromorphic controller. *Journal of Robotic Systems*, August 1988.
- [3] P. Werbos, Backpropagation through time: What it does and how to do it. *Proceedings of the IEEE*, October 1990.
- [4] D. Nguyen and B. Widrow, The truck backer-upper: an example of self-learning in neural networks. In W. T. Miller, R. Sutton, and P. Werbos, *op. cit.* [1].
- [5] P. Werbos, Maximizing long-term gas industry profits in two minutes in Lotus using neural network methods. *IEEE Trans. Systems, Man, and Cybernetics*, March/April 1989.
- [6] M. Kawato, Computational schemes and neural network models for formation and control of multijoint arm trajectory. In W. T. Miller, R. Sutton, and P. Werbos, *op. cit.* [1].
- [7] Commons, Grossberg, and Staddon, eds., *Neural Network Models of Conditioning and Action*, Erlbaum, Hillsdale, NJ, 1991.
- [8] P. Werbos, Neurocontrol and fuzzy logic: Connections and designs. *International Journal on Approximate Reasoning*, January 1992.

- [9] P. Werbos, Neurocontrol and related techniques. *Handbook of Neural Computing Applications*, A. Maren ed., Academic Press, New York, 1990.
- [10] E. Sontag, *Feedback Stabilization Using Two Hidden-Layer Nets*, SYCON-90-11. Rutgers University Center for Systems and Control, New Brunswick, NJ, October 1990.
- [11] T. Kohonen, The self-organizing map. *Proceedings of the IEEE*, September 1990.
- [12] W. Y. Huang and R. P. Lippman, Neural net and traditional classifiers. In *NIPS Proceedings 1987*.
- [13] N. DeClaris and M. Su, A novel class of neural networks with quadratic junctions. In *1991 IEEE/SMC*, IEEE Catalog No. 91CH3067-6, IEEE, New York, 1991.
- [14] R. Jacobs et al., Adaptive mixtures of local experts. *Neural Computation*, 3:(1), 1991.
- [15] S. Grossberg, Competitive learning: from interactive activation to adaptive resonance. *Cognitive Science*, 11:23–63, 1987.
- [16] P. Werbos, Advanced forecasting methods for global crisis warning and models of intelligence. *General Systems Yearbook*, 1977.
- [17] D. Sofge and D. White, Neural network based process optimization and control. In *IEEE Conference on Decision and Control* (Hawaii), IEEE, New York, 1990.
- [18] P. Werbos, Backpropagation: Past and future. In *Proceedings of the Second International Conference on Neural Networks*, IEEE, New York, 1988. (Transcript of talk and slides available from author.)
- [19] D. Shanno, Recent advances in numerical techniques for large scale optimization. In W. T. Miller, R. Sutton and P. Werbos, *op. cit.* [1].
- [20] J. E. Dennis and R. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice Hall, Englewood Cliffs, NJ, 1983.
- [21] D. Rumelhart, G. Hinton, and R. Williams, *Parallel Distributed Processing* (Chapter 8). MIT Press, Cambridge, MA, 1986.
- [22] P. Werbos, Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1:339–356, October 1988 (submitted August 1987).
- [23] M. Kuperstein, INFANT neural controller for adaptive sensory-motor coordination. *Neural Networks*, 4:(2), 1991.

- [24] M. Jordan, Generic constraints on underspecified target trajectories. In *Proceedings of IJCNN*, IEEE, New York, June 1989.
- [25] J. A. Walter, T. M. Martinez, and K. J. Schulten, Industrial robot learns visuo-motor coordination by means of neural-gas network. *Artificial Neural Networks, Vol. 1*, T. Kohonen, et al., eds., North Holland, Amsterdam, 1991.
- [26] K. Narendra and Annaswamy, *Stable Adaptive Systems*, Prentice-Hall, Englewood, NJ, 1989.
- [27] J. Johnson, Globally stable saturable learning laws. *Neural Networks*, Vol. 4, No. 1, 1991. See also A. N. Michel and J. Farrell, Associative memories via neural networks. *IEEE Control Systems Magazine*, 10:(3), 1990. Many other examples exist.
- [28] J. Von Neumann and O. Morgenstern, *Theory of Games and Economic Behavior*, Princeton University Press, Princeton, NJ, 1944.
- [29] P. Werbos, Rational approaches to identifying policy objectives. *Energy: The International Journal*, 15:(3/4), 1990.
- [30] A. Bryson and Y. Ho, *Applied Optimal Control: Optimization, Estimation and Control*, Hemisphere, 1975.
- [31] P. Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*, Ph.D. thesis, Harvard University, Committee on Applied Mathematics, Cambridge, MA, November 1974.
- [32] R. Howard, *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, MA, 1960.
- [33] P. Werbos, Consistency of HDP applied to a simple reinforcement learning problem. *Neural Networks*, 3:179–189, October 1990.
- [34] A. Barto, R. Sutton, and C. Anderson, Neuronlike elements that can solve difficult learning control problems. *IEEE Trans. Systems, Man, and Cybernetics*. 13:(5), 1983.
- [35] P. Werbos, A menu of designs for reinforcement learning over time. In W. T. Miller, R. Sutton, and P. Werbos, *op.cit.* [1].
- [36] P. Werbos, Applications of advances in nonlinear sensitivity analysis. *System Modeling and Optimization: Proceedings of the 10th IFIP Conference*, R. F. Drenick and F. Kozin, eds., Springer-Verlag, New York, 1982.

- [37] P. Werbos, Building and understanding adaptive systems: a statistical/numerical approach to factory automation and brain research. *IEEE Trans. Systems, Man, and Cybernetics*, 17:(1), January–February 1987.
- [38] P. Werbos, Neural networks for control and system identification. In *IEEE Conference on Decision and Control* (Florida), IEEE, New York, 1989.
- [39] G. Lukes, B. Thompson, and P. Werbos, Expectation driven learning with an associative memory. In *IJCNN Proceedings* (Washington), *op. cit.* [10].
- [40] C. Watkins, *Learning from Delayed Rewards*, Ph.D. thesis, Cambridge University, Cambridge, England, 1989.
- [41] T. H. Wonnacott and R. Wonnacott, *Introductory Statistics for Business and Economics*, 2nd ed., Wiley, New York, 1977.
- [42] J. Jameson, A neurocontroller based on model feedback and the adaptive heuristic critic. In *Proceedings of the IJCNN* (San Diego), IEEE, New York, June 1990.
- [43] M. Jordan and R. Jacobs, Learning to control an unstable system with forward modeling, *Advances in Neural Information Processing Systems 2*, D. Touretzky, ed., Morgan Kaufmann, San Mateo, CA, 1990.
- [44] P. Werbos and A. Pellionisz, Neurocontrol and neurobiology: New developments and connections. In *Proceedings of the International Joint Conference on Neural Networks*. IEEE, New York, 1992.

# NEURAL NETWORKS, SYSTEM IDENTIFICATION, AND CONTROL IN THE CHEMICAL PROCESS INDUSTRIES

*Paul J. Werbos*

*NSF Program Director for Neuroengineering  
Co-Director for Emerging Technologies Initiation*

*Thomas McAvoy and Ted Su*

*Department of Chemical Engineering, University of Maryland*

## 10.1. INTRODUCTION AND SUMMARY

This chapter will discuss the application of artificial neural networks (ANNs) in the chemical process industries, with particular emphasis on new designs that could be useful in other applications (neural and nonneural) as well:

- The use of simple ANNs as the system identification component of model-based predictive control (MPC), a widely used scheme that optimizes over time subject to constraints.
- The use of “robust” or “parallel” training of ANNs used in system identification. In tests on real data from a wastewater treatment plant and an oil refinery, this has led to an orders-of-magnitude reduction in prediction error when compared with conventional maximum-likelihood approaches. Earlier tests on *nonneural* implementations were also promising. There are deep theoretical lessons involved, significant to all forms of ANN adaptation and system identification.
- Techniques for pruning ANNs (a task which has turned out to be crucial to ANN performance), techniques for using ANN-like techniques with *other* nonlinear specifications (for those who do *not* want to use ANNs as such) and techniques for using ANN models similar to nonlinear ARMA models or Kalman filtering systems—to be given in later sections.

ANNs are already in use in real-world applications by many of the major chemical manufacturers. Applications fielded to date mostly involve pattern recognition or soft sensing or feedforward control



systems, where safety and stability are not a major issue. There is substantial new research, and some application, in more advanced forms of system identification and control. Greater efficiency in chemical processing, through advanced control, could save millions of dollars and have a significant impact on major global environmental problems.

This chapter will begin with a broad summary of these applications. Then, in section 10.3, it will discuss the use of simple ANNs within model-based predictive control (MPC), and discuss some tests run on a *simulated* chemical plant at the University of Maryland [1]. Section 10.4 will discuss alternative ways of adapting ANNs for use in prediction, tested against *real-world* data. The theoretical issues discussed in section 10.4 apply to system identification in general; they are not unique to ANNs. Section 10.4 will also discuss alternative model specifications—neural and non-neural—which address such issues as measurement noise, slowly-varying system parameters, and memory of important but unobserved variables. Finally, the final sections of the chapter will provide the equations needed to implement various extensions of these methods; for example, section 10.5 will discuss a technique for pruning ANNs [2], and section 10.6 will contain a tutorial on how to use the chain rule for ordered derivatives [3, 4], so as to calculate gradients and Hessians at minimum cost in a parallelizable manner for *any* nonlinear dynamic system. Sections 10.7 and 10.8 will provide equations for options discussed in sections 10.3 and 10.4.

## 10.2. GENERAL BACKGROUND ON CHEMICAL APPLICATIONS

The Chemical Engineering Department at the University of Maryland, in cooperation with the NSF-sponsored Systems Research Center, has investigated a wide range of applications of artificial neural networks (ANNs). Because this work was successful, and also because of substantial industrial interest, McAvoy founded a Neural Network Club in 1990 that now has over twenty-five dues-paying corporate members, representing Fortune 500 corporations. Many of these companies have achieved net savings already, through existing applications of ANNs, which range from hundreds of thousands to millions of dollars. At a recent public talk [5], a representative from DuPont stated that his company is already into the “third generation” of ANN applications, reaching out to dozens or even hundreds of sites within the company.

The applications of ANNs to date at the Neural Network Club are probably representative of those in the industry:

1. Sensor interpretation (biosensors) [6]
2. Nonlinear dynamic modeling [7,8]
3. Modeling (or “cloning” of human operators/experts) [9]
4. Nonlinear steady-state modeling
5. Knowledge extraction [10]
6. Smart optimization [11]
7. Fault detection [12]

In addition, a technique has been developed to remove unnecessary connections from ANNs [2] (see section 10.5), using a modified objective function. Techniques of this sort are important to the success of many applications involving backpropagation.

The sensor-interpretation example [6] is probably typical of the easy, first-generation applications that are now widespread and still expanding. In this example, McAvoy et al. were basically trying to build a "soft sensor" capable of monitoring the concentration of desirable products in real time in a chemical plant. Traditionally, the process industries have relied on simple measurements such as flow, pressure, and temperature; however, to achieve a higher level of efficiency, and to control more complex systems such as bioreactors, it is important to measure concentrations as well.

Chemical engineers have known for decades how to measure chemical concentrations *in the laboratory*, if one is willing to wait for a few hours or a few days. The challenge here lies in *real-time* sensing. We do have real-time measurements available—such as fluorescence spectra—which *implicitly* tell us what the concentrations are; the problem is to learn the nonlinear mapping from the *spectral information* to *actual concentrations*. In their experiment, McAvoy et al. did exactly that, by putting together thirty-three samples of various mixtures of amino acids, and using a simple ANN to predict the corresponding concentrations of fluorophores. They achieved an average absolute prediction error of 5.2% in predicting the mole fractions, using an ANN, versus 11.7% when they used partial least squares (PLS), the best conventional alternative [13]. A larger training database should make the method more accurate.

ANNs for pattern recognition have also been used to detect hot spots in the continuous casting of steel [14]. In this application, ANNs led to a great increase in accuracy over more expensive, conventional methods. The system has been installed in several plants of the Nippon Steel Company and is scheduled for use in all of them.

Nonlinear dynamic modeling—like smart sensing—is crucial to real-time control. For that reason, this chapter will describe our recent experience in that area in some detail. First-principle models are possible in the chemical industry, as in other industries; however, the sheer complexity of chemical plants and the presence of unknown secondary reactions usually make such models extremely expensive, unreliable, and specific to only one or two plants. For this reason, *empirically* based models—either linear models or ANN models—can be of great value. ANN models can be used either as alternatives to first-principle models or as supplements, to predict the errors of the first-principle models.

The modeling of human operators can also be very useful [9]. The difference between the best human operators and the average is often worth large amounts of money, because of differences in plant efficiency during plant change-over and the like. The techniques required to model human operators with ANNs are no different, in principle, from those required to model other dynamic systems. This approach is discussed further in the section on supervised control in Chapter 3.

ANNs have been used at Maryland and elsewhere to extract knowledge that is then used in more conventional AI control systems [10]. There are probably dozens of ways to combine ANNs and AI techniques, useful in different applications, especially if the AI systems are based on differentiable functions (as in fuzzy logic). Many of the best examples of such combinations come from Japan [15].

More effective optimization—both in the design of chemical plants [11] and in real-time optimization—will probably turn out to be the most important application in the long term. As an example, imagine a petrochemical plant that produces a billion dollars worth of product per year, running at a ninety-eight percent level of efficiency. Such efficiencies are quite typical. The remaining *inefficient*

cies are typically due to the difficulty of optimization during transient or changing conditions, when process parameters change in an unplanned or nonlinear manner. Good nonlinear optimization might well raise this up to ninety-nine percent efficiency, thereby saving the company \$10 million per year.

More importantly, a rise in efficiency from ninety-eight percent to ninety-nine percent typically cuts the *unwanted waste products* in half, from two percent to one percent. It is not obvious what the larger social implications of this may be, because there are so many different kinds of plants and waste products; however, as a general rule, there *is* a correlation between inefficiency and waste, and there *is* reason to believe that such improvements could have a substantial impact on environmental problems. As a further example, a representative of General Electric (GE) has told us that more efficient, pattern-recognition-based control systems might well be able to reduce the emissions of NO<sub>x</sub> by a factor of ten, in gas turbines and boilers, even as a retrofit to well-designed modern systems. These kinds of environmental benefits address central problems that would otherwise be very difficult to deal with in a comprehensive, long-term energy and environmental policy [55].

Better control may be crucial, as well, to the *feasibility* of using certain advanced production processes, such as the biological processes used in biotechnology and wastewater treatment plants. The potential markets for such processes are very large.

Better diagnostics are important, of course, to improved safety and to cost-effective maintenance that can minimize the need for very expensive replacement investments. A wide variety of techniques has been used at the University of Maryland and elsewhere [12,14,16]. The issue of safety will be a crucial factor in the use of novel control designs in the chemical industry.

### 10.3. MODEL-BASED PREDICTIVE CONTROL WITH ANNS

Model-based predictive control (MPC) has been used very widely in the chemical and petroleum industries. Because of the difficulties of using or obtaining first-principle models (mentioned above), the usual procedure is to develop a *linear* model of the plant, based on empirical data, and to use that model within an optimization routine. Our goal has been to use the same empirical data, instead, to develop an ANN model of the same plant. This is almost as easy as developing the linear model; however, because ANNs can learn to represent any well-behaved nonlinear function [17,18,19], they should be able to represent the plant more accurately, and thereby lead to better performance.

The simulations described here were first reported by Saint-Donat, Bhat, and McAvoy [1] at Maryland. Saint-Donat has since returned to the ATOCHEM company of France, and Bhat has gone to Texaco.

Model-based control systems have been used before in the neural network community. For example, see the section on the “backpropagation of utility” in Chapters 3 and 13. The method used by Saint-Donat et al. is technically a form of the backpropagation of utility, but it has two special features that it shares with other forms of MPC:

- The present and future control signals are output *directly* from the optimizer, not from an ANN or other parametrized controller; in other words, the system solves for an optimal future schedule of actions, rather than an optimal set of controller parameters.
- The optimizer takes account, explicitly, of system constraints, which typically involve constraints on flow rates, temperature, and pressure.

This method is parallelizable, and the major pieces of it could be implemented on chips; therefore, unlike traditional MPC, it has serious promise as a real-time optimal control system.

This section will begin with a more detailed description of the task that Saint-Donat et al. addressed in simulation. Next, it will describe the architecture used to perform the task. It will conclude with a discussion of the observed performance.

### 10.3.1. The Task: A pH CSTR Reactor

The task is to control the level of acidity (pH) in a continuously stirred tank reactor (CSTR), illustrated in Figure 10.1. We assume a constant input flow of acetic acid. Our goal is to adjust the flow of sodium hydroxide (NaOH) so as to “neutralize” the acid—to achieve a desired setpoint for the pH. To test the control system, Saint-Donat et al. evaluated its ability to cope with a change in setpoints. In the initial tests, they changed the setpoint from 6.5 to 6.3. In later tests, they changed it from 7.2 to 7.0.

At any given time,  $t$ , we have information on the pH (which we observe) and the flow rate (which we chose), both for times  $t - 3$ ,  $t - 2$ , and  $t - 1$ . We also observe the pH at time  $t$ . Our goal is to decide on the flow rate for times  $t$  through  $t + 7$ . The flow rates for times  $t + 8$  through  $t + 30$  will be set equal to the flow rate for time  $t + 7$ ; this is done to stabilize the system, to prevent “overoptimization,” and is standard in MPC.

To be more precise, our goal at any time  $t$  is to pick the actions ( $F_2(t)$  through  $F_2(t + 5)$ ), which will minimize the tracking error:

$$U = \sum_{k=1}^{30} (pH(t+k) - pH^*(t+k))^2, \quad (1)$$

where  $pH$  is the actual pH and  $pH^*$  is the desired pH, subject to four constraints that apply at all times:

$$pH_{\min} \leq pH \leq pH_{\max} \quad (2)$$

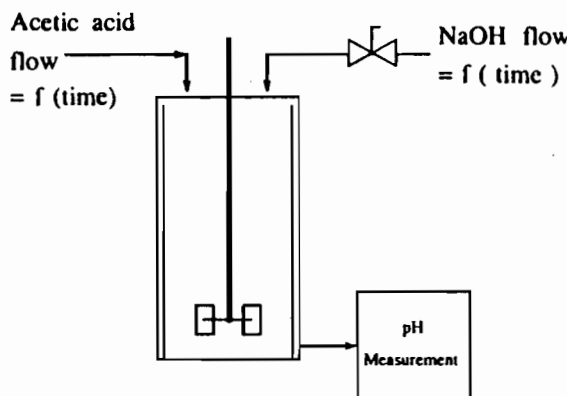


Figure 10.1 Case study (pH in a stirred tank reactor)

$$F_{2\min} \leq F_2 \leq F_{2\max}. \quad (3)$$

Tracking error is denoted by "U," rather than "E," because it is really a measure of utility or disutility—a measure of the desirability of the actual state of the plant; the letter "E" is reserved for prediction errors or classification errors or the like. Our control system as a whole is designed to minimize U, but it will contain a system identification component trained to minimize E.

The simulation was based on equations taken from [7]. By writing material balances on  $\text{Na}^+$  and total acetate ( $\text{HAC} + \text{AC}^-$ ), and assuming that acid-base equilibria and electroneutrality relationships hold, one gets:

*Total Acetate Balance:*

$$\xi = [\text{HAC}] + [\text{AC}^-] \quad (4)$$

$$V \frac{d\xi}{dt} = F_1 C_1 - (F_1 + F_2) \xi \quad (5)$$

*Sodium Ion Balance:*

$$\zeta = [\text{NA}^+] \quad (6)$$

$$V \frac{d\zeta}{dt} = F_2 C_2 - (F_1 + F_2) \zeta \quad (7)$$

*Electroneutrality:*

$$\zeta + [\text{H}^-] = K_w / [\text{H}^+] + K_a [\text{HAC}] / [\text{H}^+] \quad (8)$$

$F_1$  and  $F_2$  are the acid and base flow rates, respectively. In our case,  $F_2$  is the manipulated variable.  $C_1$  and  $C_2$  are the concentrations of the acids and bases, respectively, in the input stream.  $V$  is the volume of the reactor. The parameters used in this simulation are given in Table 10.1.

**Table 10.1**

CSTR Parameters Used	Value
Volume of Tank	1000 lit.
Flow rate of HAC	81 lit./min.
Steady state flow rate of NaOH	515 lit./min.
Steady state pH	7
Concentration of HAC in F1	0.3178 mol./lit.
Concentration of NaOH in F2	0.05 mol./lit.

## MPC APPROACH

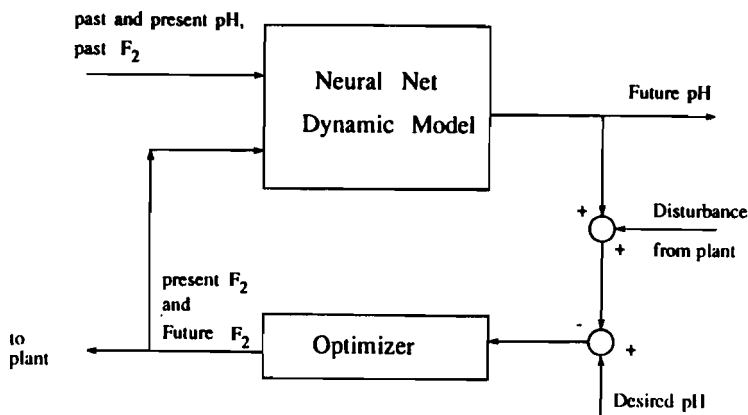


Figure 10.2 Neural net based control.

## 10.3.2. Architecture Used to Solve the Problem

The architecture used to solve this problem is shown in Figure 10.2. First of all, an ANN was adapted or fitted to serve as a dynamic model of the process to be controlled. Then, in real-time control, the ANN was held fixed, except for a constant bias term that was continually updated. At each time  $t$ , the optimizer looked ahead thirty steps and planned a series of actions for thirty time-steps ahead; however, it actually took only one step, and then—at time  $t + 1$ —it recalculated everything. This section will describe these steps in more detail.

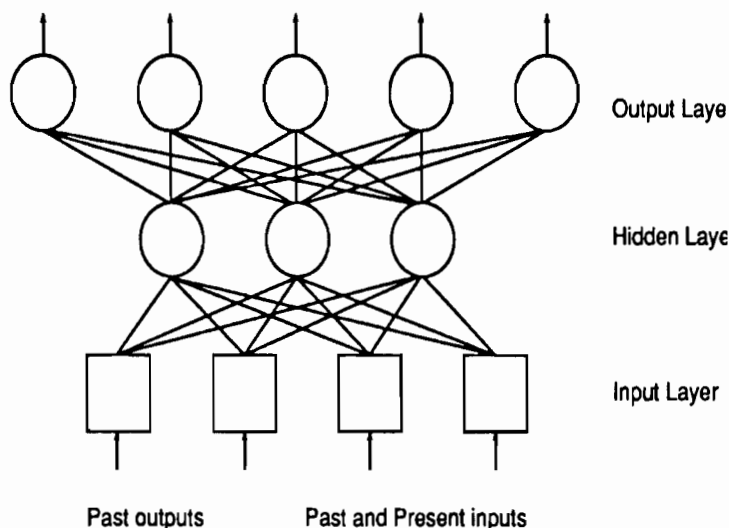
**10.3.2.1. Adapting the ANN** The ANN was an ordinary three-layer multilayer perceptron (MLP), illustrated in Figure 10.3. In general, this kind of network inputs a vector of inputs,  $X(t)$ , made up of components  $X_1(t)$  through  $X_m(t)$ . It outputs a vector  $Y(t)$ , made up of components  $Y_1(t)$  through  $Y_n(t)$ . It obeys the following system of equations at each time  $t$ :

$$v_j^- = W_{j0}^- + \sum_{k=1}^m W_{jk}^- X_k(t) \quad 1 \leq j \leq h \quad (9)$$

$$x_j^- = s(v_j^-) \triangleq 1/(1 + e^{-v_j^-}) \quad 1 \leq j \leq h \quad (10)$$

$$v_i^+ = W_{i0}^+ + \sum_{j=1}^h W_{ij}^+ x_j^- \quad 1 \leq i \leq n \quad (11)$$

$$\hat{Y}_i(t) = x_i^+ = s(v_i^+) \triangleq 1/(1 + e^{-v_i^+}), \quad 1 \leq i \leq n \quad (12)$$



**Figure 10.3** Multilayer perceptron.

where  $h$  represents the size of the hidden layer, where  $s()$  is usually called “the sigmoid function,” where the matrices  $W^*$  and  $W$  are the “weights” or parameters to be adapted, where  $v^-$  and  $v^+$  represent the levels of voltage stimulating the neurons on the hidden and output layers, respectively, and where  $x^-$  and  $x^+$  represent the intensity of the outputs of those neurons.

In this application, Saint-Donat et al. used a net with seven inputs (i.e.,  $m = 7$ ), four hidden units ( $h = 4$ ), and one output or target variable ( $n = 1$ ). At each time  $t$ , the input vector,  $X(t)$ , consists of the following seven numbers:  $pH(t-3)$ ,  $pH(t-2)$ ,  $pH(t-1)$ ,  $F_2(t-3)$ ,  $F_2(t-2)$ ,  $F_2(t-1)$ , and  $F_2(t)$ . The output,  $Y_t$ , was used to predict  $pH(t)$ . In actuality, the variable  $F_2(t)$  had little effect on the forecast of  $pH(t)$ , because of deadtime effects, and was deleted from subsequent tests; however, this chapter will assume its inclusion, for the sake of generality. In earlier experiments [7], ANNs had been developed to predict  $pH(t+1)$ ,  $pH(t+2)$ , and  $pH(t+3)$  all together as outputs of the network; however, those ANNs were not used in this application. Figure 10.5 illustrates the fit between predicted pH and actual pH in some of the earlier work [7].

In order to adapt this network—i.e., to estimate the optimal values of the weights  $W_{ij}^-$  and  $W_{ij}^+$ —Saint-Donat, et al. performed two tasks: (1) they developed a database of simulated  $X(t)$  and  $pH(t)$  for many times  $t$ ; (2) they used the well-known technique called backpropagation to adapt the weights, so as to minimize the error of the network in predicting  $pH(t)$  over that database.

The technique for developing the database was taken from Bhat, et al. [7]. Basically, a pseudo-random binary signal (PRBS) was added to a steady-state signal for the manipulated variable, and equations 4 through 8 were used to forecast the resulting flows. This led to the time series for flow rate ( $F_2$ ) and  $pH$  shown in Figure 10.4. This continuous time series was read off at regular time intervals, yielding the database for  $X(t)$  and  $pH(t)$  across different values of  $t$ .

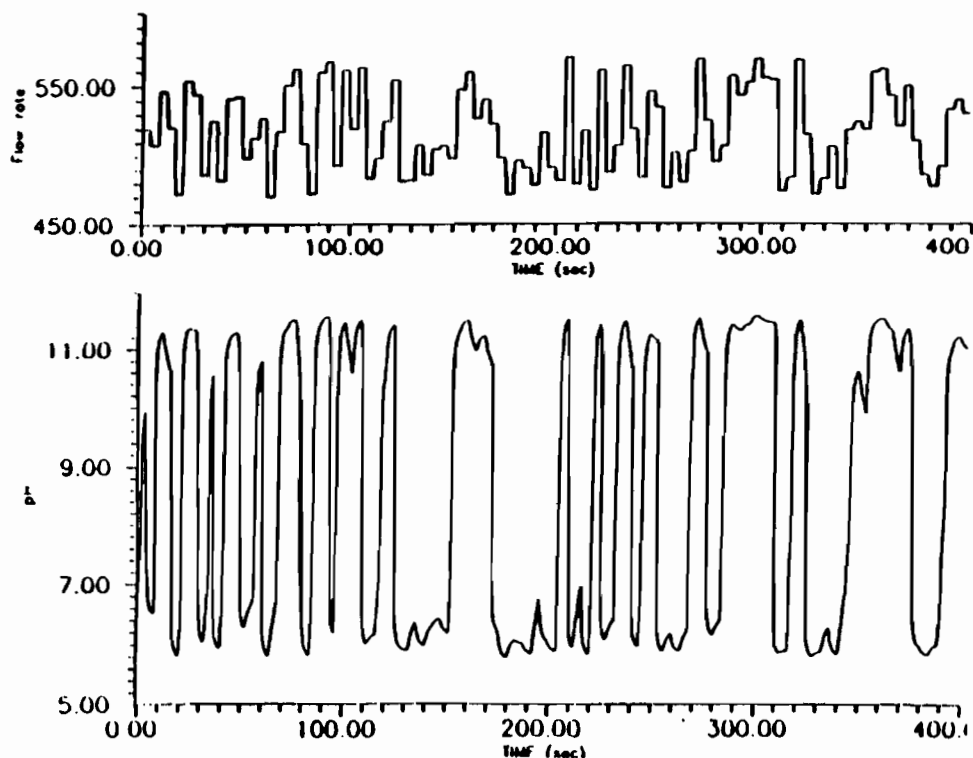


Figure 10.4 Case study: pH in a C.S.T.R. (Acid flow rate constant and NaOH flow rate varying.)

The technique used to adapt the weights was basic backpropagation (BEP)—a specialized use of backpropagation first proposed by Werbos [20–22] and later popularized by Rumelhart [23]. In this application, Saint-Donat et al. cycled through each of the  $(X(t), pH(t))$  pairs in the database, performing the following sequence of steps for each pair  $t$ :

1. Use equations 9 through 12 to calculate a forecast,  $\hat{Y}(t)$ , for  $pH(t)$ . We will denote this forecast as  $p\hat{H}(t)$ .
2. Calculate the error of this forecast:

$$E(t) = 1/2(p\hat{H}(t) - pH(t))^2 \quad (13)$$

3. Update the weights by the equations:

$$\text{new } W_{ij}^+ = \text{old } W_{ij}^+ - \text{learning\_rate} * \frac{\partial E(t)}{\partial W_{ij}^+} \quad (14)$$



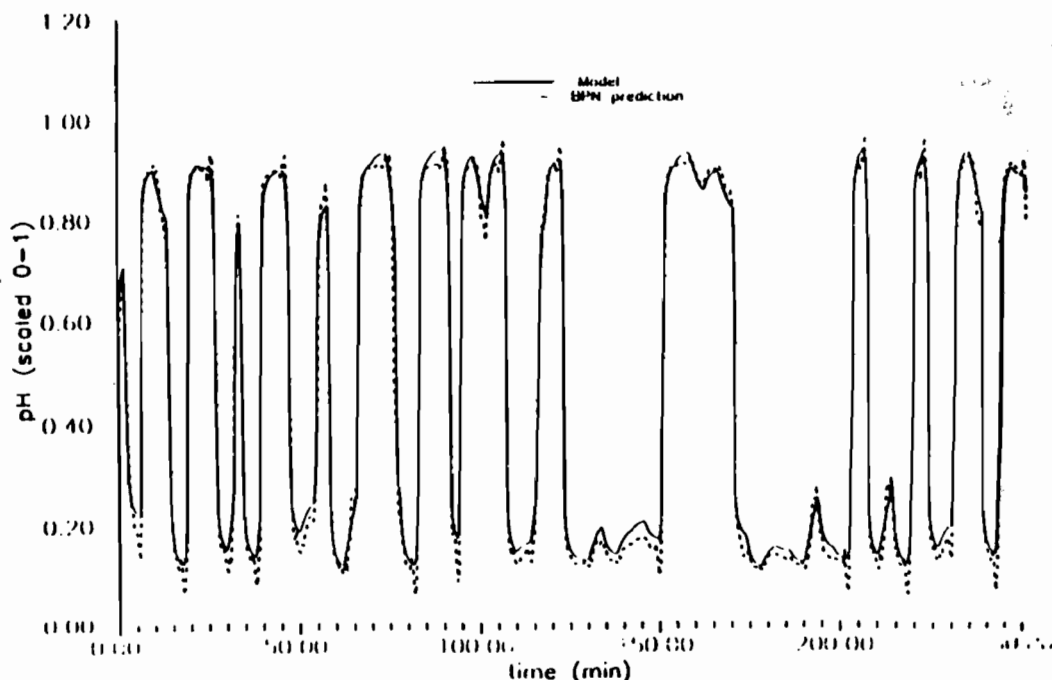


Figure 10.5 Prediction on training data (st. st. pH = 9).

$$\text{new } W_{ij} = \text{old } W_{ij} - \text{learning\_rate} * \frac{\partial E(t)}{\partial W_{ij}},$$

where the constant *learning\_rate* was set as in [7]. They cycled through all of the  $(X(t), pH(t))$  pairs, over and over, until the weights converged.

For the simple architecture shown in equations 9 through 12, it happens that:

$$\frac{\partial E(t)}{\partial W_{ij}^*} = (\hat{Y}_i(t) - Y_i(t)) * s'(v_i^+) * x_j^- \quad (15)$$

and

$$\frac{\partial E(t)}{\partial W_{jk}^*} = \left( \sum_{i=1}^n (\hat{Y}_i(t) - Y_i(t)) * s'(v_i^+) \right) * W_{ij}^* s'(v_j^-) X_k(t), \quad (16)$$

where the sigmoid function  $s(\cdot)$  has the property:

$$s'(v) = s(v)(1 - s(v)). \quad (17)$$

In this application,  $n = 1$  and  $Y_1(t)$  is just  $pH(t)$ ; this simplifies the calculations considerably.

Backpropagation, in the most general sense, refers to a general algorithm for calculating derivatives efficiently through a large sparse system, to be described in section 10.6. The efficient implementation of equations 16 and 17 is simply one example of the applications of that algorithm. Many authors have discussed alternatives to equation 14 which can speed up convergence [21,24,25]

**10.3.2.2. Adapting the ANN Bias in Real Time** Once the ANN was adapted off-line, as discussed in section 10.3.2.1., the weights were frozen *throughout* the control simulations, with one exception. At each time  $t$ , the forecast of  $pH(t)$  actually used was:

$$pH(t) \text{ forecast} = p\hat{H}(t) + \text{bias}, \quad (18)$$

where  $p\hat{H}(t)$  was the forecast from the ANN and *bias* was calculated from the previous time-step:

$$\text{bias} = pH(t-1) - p\hat{H}(t-1). \quad (19)$$

Furthermore, this same value of *bias* was used in predicting  $pH(t+k)$ , for  $k = 1$  through 30, as required by the control system (optimizer) at time  $t$ . The bias term accounts for plant/model mismatch and unmeasured disturbances. By using equation 19, it can be shown that perfect setpoint tracking can be achieved in spite of these effects.

**10.3.2.3. Solving the Optimization Problem in Real Time** At each time  $t$ , the optimization routine solved for values of  $F_2(t)$  through  $F_2(t+7)$  to maximize:

$$U = \sum_{k=1}^{30} (pH(t+k) - pH^*(t+k))^2,$$

subject to the constraints:

$$\begin{aligned} pH_{\min} &\leq pH \leq pH_{\max} \\ F_{2\min} &\leq F_2 \leq F_{2\max} \\ F_2(t+k) &= F_2(t+7) \quad \text{for } k > 7, \end{aligned}$$

as given in equations 1 through 3, assuming that  $pH(t+k)$  is given as in equation 18.

Actually, equation 18 only tells us how to predict  $pH(t)$  when we know  $F_2(t)$  and prior information; Figure 10.6 shows the general approach used to "chain" the ANN to produce all the forecasts we need, from  $pH(t)$  through to  $pH(t+30)$ . (Some of these figures have minor differences from the nets used here, because they came from earlier work.)

This optimization problem is a straightforward problem in nonlinear programming. To solve this problem, Saint-Donat et al. [1] used a feasible sequential quadratic programming (FSQP) program developed through research at the University of Maryland [26,27]. This program can:

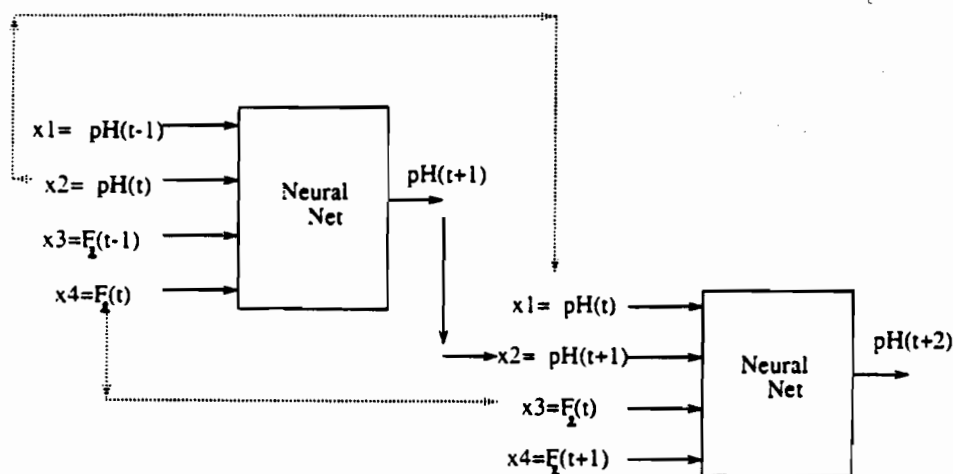


Figure 10.6 Simple example.

- Minimize any smooth nonlinear function
- Deal with any kind of nonlinear equality and/or inequality constraints on the manipulated variables
- Generate only feasible iterates (i.e., meet all the constraints even before convergence to the final solution)

This last feature will be especially important if we actually try to apply this approach in a real plant. In a real plant, one can never be certain that the FSQP will find the optimal solution quickly enough, if we require it to stop its calculations at the end of each sampling cycle. This could become a serious problem for complex plants. However, the feasibility property guarantees that the current iteration of FSQP will still give us acceptable values for the manipulated variables, even if they are not quite fully optimal. The feasibility feature has another advantage, which may help in speeding up convergence: By keeping the iteration out of the unacceptable range, it also keeps it within the original training range of the ANN; iterations outside that range might have confused the iterative process, because predictions from the ANN are far less reliable outside of the original training range.

In the future, there is reason to hope that this algorithm can be implemented in a fully parallel form, on a chip—enabling true real-time applications. ANN calculations can certainly be done on a chip. Even within the optimization algorithm, a major part of the computation concerns the gradient of the Hamiltonian function for the problem, given by:

$$\begin{aligned}
 H = & U + \sum_{k=0}^{30} \left( \lambda_k^{(1)} (pH(t+k) - pH_{\min}) + \lambda_k^{(2)} (pH_{\max} - pH(t+k)) \right) \\
 & + \sum_{k=0}^{30} \left( \lambda_k^{(3)} (F_2(t+k) - F_{2\min}) + \lambda_k^{(4)} (F_{2\max} - F_2(t+k)) \right)
 \end{aligned} \quad (20)$$

where the  $\lambda$ 's are Lagrange multipliers. In the special case where  $pH(t+k)$  is predicted by an ANN, we can parallelize the calculation of this gradient, simply by using backpropagation through time. This was actually done successfully in the simulations by Saint-Donat et al. In fact, backpropagation can also be used to calculate the Hessian of the Hamiltonian—which can also be used in FSQP—in a highly efficient way. section 10.7 will provide the details of how to do both things, based directly on the chain rule for ordered derivatives.

Additional speedup results from using the values of  $F_2(t+1)$  through  $F_2(t+30)$ , calculated at time  $t$ , as *starting* values for the optimizer at time  $t+1$ . Deeper analysis of these Lagrange multipliers may also be useful. (However, users of methods like differential dynamic programming or DHP should be warned that these Lagrange multipliers are quite different, because the equations used to predict  $pH(t+k)$  are not treated as constraints themselves.)

### 10.3.3. Simulation Results and Conclusions

Before testing the overall control scheme, Saint-Donat et al. conducted a brief test of the steady-state predictive abilities of the ANN. Figure 10.7 shows a comparison of the steady-state pH versus the steady-state flow of NaOH into the CSTR for a constant inflow of acid. As can be seen, the ANN achieves a reasonably accurate model of the highly nonlinear steady-state characteristics of the CSTR.

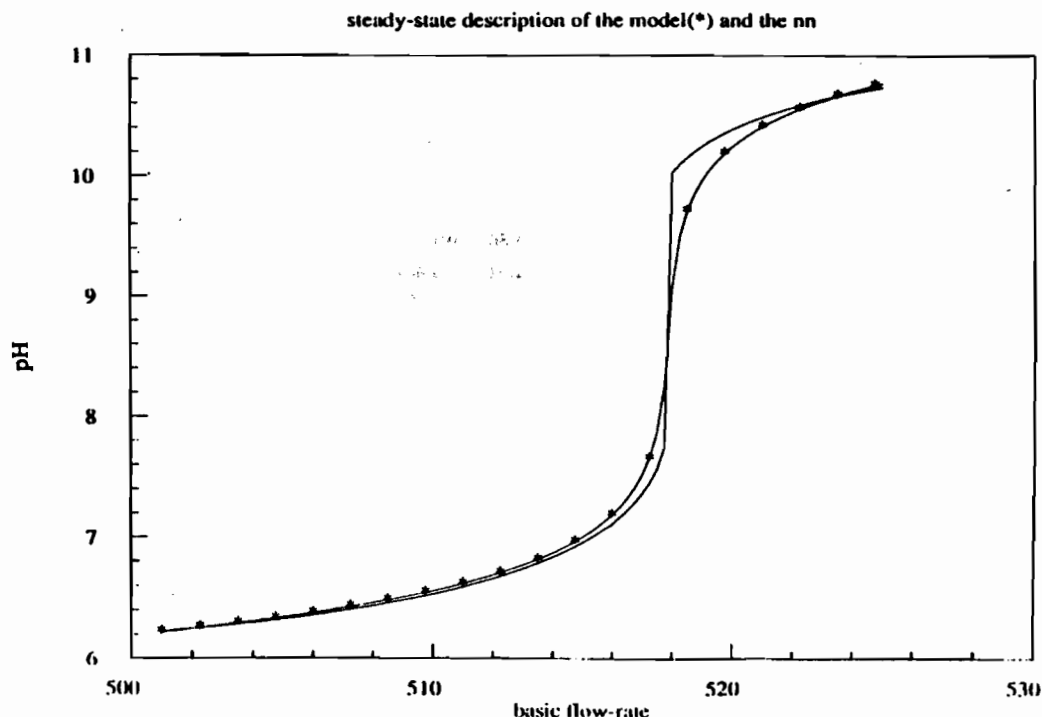
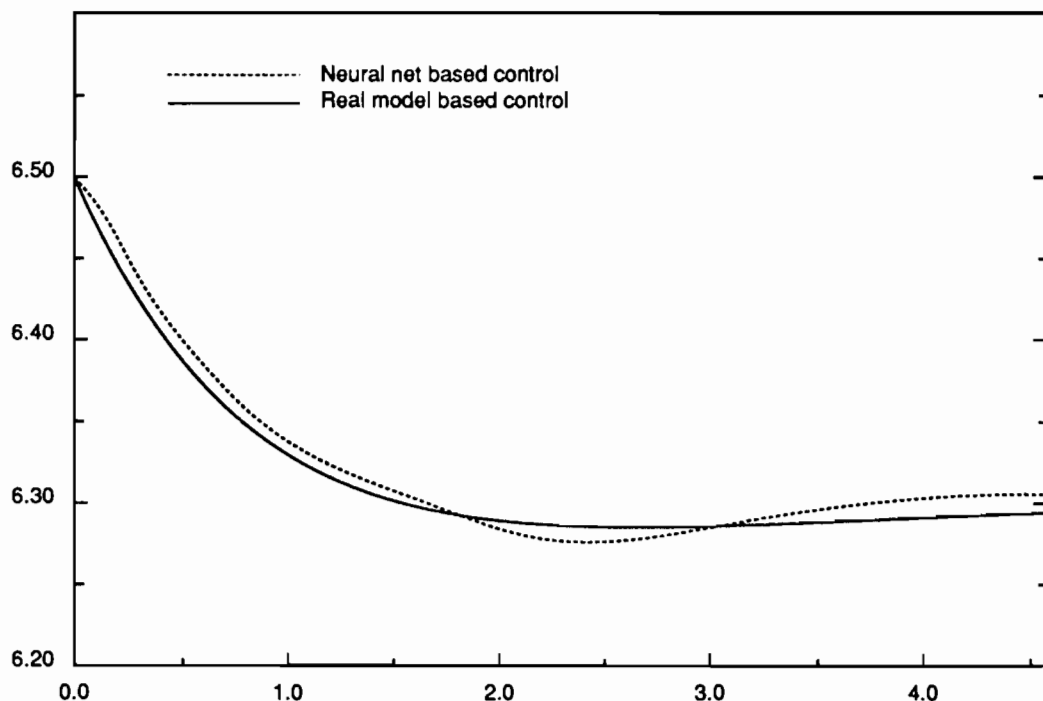


Figure 10.7 Steady state description of the model(\*) and the nn.



**Figure 10.8** Controlled set-point change: 6.5  $\rightarrow$  6.3.

To test the model-based control approach, the pH setpoint was initially set to 6.50. Then the setpoint was changed to 6.3. Figure 10.8 shows the actual change in acidity achieved by the neural net control system. Also shown is the response achieved when the real model is used in place of the ANN in Figure 10.2. As can be seen, the response achieved with the ANN is close to that achieved using the real model. In both cases, the new setpoint is not reached immediately, because the optimization hits constraints on the allowable flow. It is clear in this case that the ANN has captured the dynamics of the process. There was no need for first-principle modeling; the control architecture shown in Figure 10.2 does not need any prior knowledge about the plant. The ANN, although trained on a very wide region, is able to predict very accurately the dynamics of the pH system in this region between 6.5 and 6.3.

To provide a more challenging problem, the pH setpoint was initially set at 7.2. Then the setpoint was changed to 7.0. In this pH region, the steady-state pH gain to basic flow (i.e., the derivative of pH with respect to flow) changes by a factor of 2.8, as Figure 10.7 illustrates. Figure 10.9 shows the setpoint responses achieved by the ANN control system. Also shown is the response achieved when the real model is used in place of the ANN in Figure 10.2. As can be seen, the ANN-controlled response is slower than that of the real model and its undershoot is higher. It should be noted that the ANN was trained on a database where the pH varied between 6 and 11. If the ANN were trained on

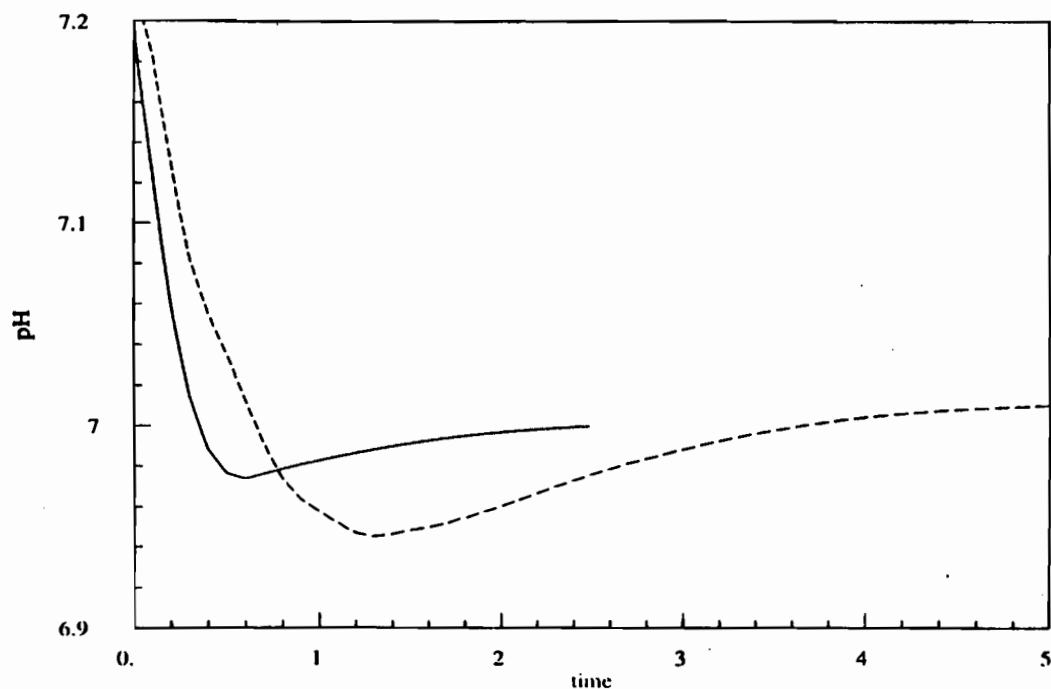
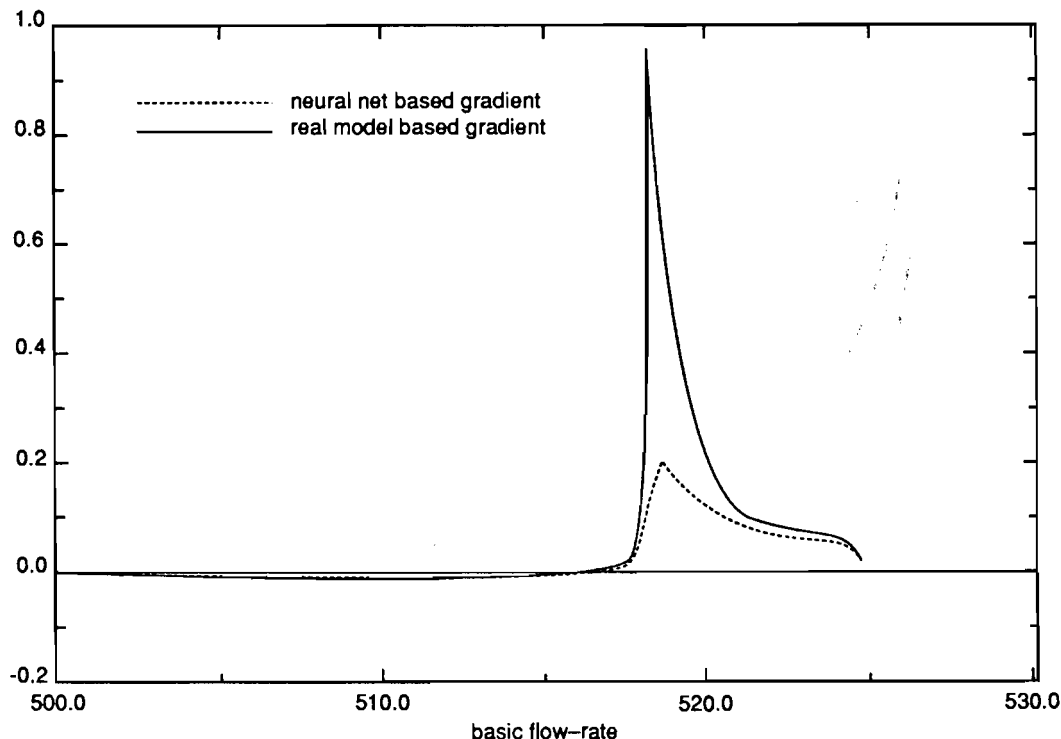


Figure 10.9 Controlled set-point change: 7.2  $\rightarrow$  7.0.

data in the pH range 7.0 to 7.2, a more accurate model would be achieved. (This also suggests the possibility of something like neural gain scheduling, as a step up from the usual linear gain scheduling.) Although the process gain changes by a factor of 2.8 in the region between pH equal 7.2 to 7.0, the neural net is able to cope with this change effectively. Overall, the results shown in Figure 10.9 are encouraging, since the nonlinear neural net model was developed using only process data.

As can be seen, the ANN model used in a model predictive control architecture produces a response in certain regions that is slower than that produced by the real model. The question arises as to how this discrepancy can be explained. Figure 10.10 shows the derivative of the tracking error  $U$  with respect to the first future move of the manipulated variable ( $F_2$ ). For each flow rate on the X-axis, the steady state is assumed to be reached. Tracking errors were calculated based on a setpoint of 7.0, which corresponds to an NaOH flow of 515 liters per minute. The ANN does a good job in most of the cases but has some problems in modeling the dynamics near pH = 7. This case study is particularly nonlinear, with the classic pH jump near 7. Given that the control architecture is tested in this region of pH near 7, results like those in Figures 10.9 and 10.10 are easy to explain. These results do not represent the limit of our system to control a plant, but rather the limits of this ANN in modeling velocity information. The ANN was trained on real input-output data, and was asked after convergence, not only to reproduce that data, but also to provide the first and eventually the second derivative



**Figure 10.10** First gradient of the objective function.

information, on which it hasn't been trained. (It is possible, in principle, to augment the error function  $E(t)$  by adding the length of the difference between the *gradient* vectors of the actual model and the ANN. The adaptation is straightforward and inexpensive, using second derivative techniques analogous to those used in "GDHP" [28]; however, actual gradient vectors are not generally available for real plants.) Perhaps such derivative information would be more accurate if narrower training regions were used, or if a larger network were used (with a larger training set). Other traditional chemical engineering processes should not be so demanding in terms of their nonlinear behavior compared to the pH system studied in this section.

#### 10.4. TOWARDS A MORE ROBUST TRAINING OR IDENTIFICATION PROCEDURE: EMPIRICAL RESULTS, ALTERNATIVE MODELS, AND THEORY

The previous section has shown how the quality of system identification with an ANN can have a strong effect on the resulting quality of control. Chapters 3 and 13 argue that the quality of system identification is crucial to the capabilities of neurocontrol in general, even when other designs are

used. Most of the failures of neurocontrol in the past year or two can be traced back to inadequate system identification. This section will discuss alternative ways of adapting ANNs for the purpose of prediction, which appear to work far better than the conventional methods described in section 10.3.

The problem of system identification is not unique to neural networks. In the control theory community as well, a variety of techniques have been tried. Linear system identification has made great progress, and many generic linear models, such as transfer-function models and convolution models, are available and widely used. For example, transfer function models have been used as fundamental tools for the design and analysis of control systems. Convolution models, such as a finite impulse response model (FIR) or a step response model, have played an essential role in the dynamic matrix control algorithm (DMC). However, linear models can only be used under limited conditions and special assumptions, because most physical processes usually behave nonlinearly.

This section will begin by discussing new empirical results, obtained with an alternative training procedure in the *nonlinear* case [29]. The ANN that we adapted was a very simple static network, as in section 10.3. First we will describe how we adapted the network. Next, we will describe the real-world data and results, first for the wastewater treatment plant and then for the oil refinery. Next, in section 10.4.5, we will compare our simple ANN model and design against similar designs—such as ARMA models and their ANN generalizations—which could lead to better performance in more demanding applications. Finally, in section 10.4.6, we will describe the deeper statistical issues that underlie our training procedure, along with the *nonneural* tests of similar methods and their extensions. We will describe some alternative training methods that are likely to be more powerful than what we tested here. The issues here are extremely subtle, and there is clearly room for further research to do even better. Section 10.8 will provide some of the equations needed to implement these models and methods.

*The reader should be warned that the methods described in the first part of this section will work very poorly in some situations.* That is the reason why section 10.4.6 is important for some applications and why more research is needed. In some applications, however, the simple methods of section 10.4.1 do lead to an orders-of-magnitude reduction in prediction errors.

Even before we discuss our empirical results, we should say a few words about our use of the word “robust.” In control theory, “robust control” typically refers to the design of linear controllers that are “robust” in the sense that they achieve stability (if not optimality) in control even as the parameters of the unknown linear plant vary over a wide interval. This is merely one special case of “robustness”—the effort to build systems that work well in practice *despite* our lack of perfect information about the plant to be controlled *or* predicted. The phrase “robust estimation” was used long ago in the field of statistics [30], to refer to identification procedures that tend to yield good predictions despite our lack of knowledge about the equations governing the real world. The effort described here was motivated by earlier work on robust estimation [3]. Again, the theory involved is quite subtle and will be described in more detail in section 10.4.6.

### 10.4.1. Networks and Procedures Used in System Identification in Our Tests

**10.4.1.1. Structure of the ANNs** In all these tests, we used a simple MLP, as described in equations 9 through 12. However, the vectors  $Y(t)$  to be predicted had more components here. In the three wastewater treatment tests, we tried to predict the concentrations of three ions coming out of



the various treatment subsystems:  $\text{NH}_4^+$ ,  $\text{NO}_3^{2-}$ , and  $\text{PO}_4^{3-}$ ; thus,  $Y(t)$  had three components, corresponding to these three concentrations, in each of these tests. In the oil refinery test, we tried to predict two production variables that we cannot describe because of the proprietary nature of the data; thus  $Y(t)$  had two components. In all of these tests, the *inputs* to each network included data on other variables ( $u$ ) from earlier times. The inputs *also* included  $Y(t-1)$  in all cases. In two of the four tests, we also supplied  $Y(t-2)$  and  $Y(t-3)$  as inputs to the network. The approach we used here would *not* be useful in cases where  $Y(t-1)$  is unimportant as an input in predicting  $Y(t)$ ; however, cases like that are relatively rare in control applications.

**10.4.1.2. Three Alternative Approaches to System Identification** This section will only describe the two approaches tested here, and a related approach introduced for the sake of explanation. See section 10.4.6 for other approaches.

In all four tests, we used an ANN to predict  $Y(t)$  as a function of earlier data. We may call this vector-valued function " $F$ ," and we may write our model as:

$$\hat{Y}(t) = F(Y(t-1), Y(t-2), Y(t-3), u(t-1), u(t-2), u(t-3), W), \quad (21)$$

where  $F$  is the function that represents the neural network, and  $W$  is the set of weights we are trying to estimate/identify/adapt.

The most common way to estimate/identify/adapt a function of this kind is to pick  $W$  so as to minimize:

$$E = \frac{1}{2} \sum_{ij} (Y_i(t) - \hat{Y}_i(t))^2. \quad (22)$$

Many alternative error functions (such as the 1.5 power of error) have been used for systems that do not have Gaussian noise distributions [30]; however, that aspect of robust estimation is not our major concern here. (Also, it is trivial to adapt our approach to such alternatives.) In our case, we will always minimize error as defined in equation 22, but we will minimize three different measures of error based on three different ways of interpreting equation 22. The meaning of equation 22 is not well specified until we specify how to calculate " $\hat{Y}_i(t)$ " as a function of  $W$  and of our data. The most common procedure, based on maximum likelihood theory and a simple model of noise, is to substitute equation 21 into equation 22 and minimize the following expression as a function of  $W$ :

$$E_1 = \frac{1}{2} \sum_{ij} (Y_i(t) - F_i(Y(t-1), Y(t-2), Y(t-3), u(t-1), u(t-2), u(t-3), W))^2 \quad (23)$$

where  $Y(t-\dots)$  refers to actual observed data for  $Y$ , and where  $F_i$  refers to the  $i$ th component of the function  $F$ . This is the procedure we used in section 10.3, and it is also the procedure used in the bulk of ANN research.

An alternative approach is to build up prediction for  $Y(t)$  from *predicted* values in earlier time periods. Thus, we may calculate a prediction:

$$\hat{Y}(t) = F(\hat{Y}(t-1), \hat{Y}(t-2), \hat{Y}(t-3), u(t-1), u(t-2), u(t-3), W) \quad (24)$$

and therefore try to minimize the following function of  $W$ , treating the other arguments of the function as constants:

$$E_2 = \frac{1}{2} \sum_{i,j} (\hat{Y}_i(t) - F_i(\hat{Y}(t-1), \hat{Y}(t-2), \hat{Y}(t-3), u(t-1), u(t-2), u(t-3), W))^2. \quad (25)$$

Approaches of this sort have been used quite often in the linear case in control theory, particularly by Ljung and by Narendra [31–33]. Usually they are called parallel identification methods.

Notice that equations 23 and 25 are *both* defined, at time  $t$ , as functions of a small number of inputs available at time  $t$  (and earlier). Therefore, both equations would allow us to adapt an ANN on an observation-by-observation basis, as in section 10.3. Both are suitable for *real-time* system identification. In these tests, however, we studied the problem of *off-line* system identification.

In off-line system identification, it is possible to do something very similar to equation 25, while also accounting for the influence of the weights  $W$  in *changing* the values of  $\hat{Y}(t-1)$ ,  $\hat{Y}(t-2)$ , and  $\hat{Y}(t-3)$ , as they are input to the neural net. To explain this idea in formal terms, we need to consider how equation 24—chained back from time  $t$  to time 4 (where time  $t=1$  is defined as the earliest data point)—leads to the functional relationships:

$$\begin{aligned} \hat{Y}(t) &= F(\hat{Y}(t-1), \hat{Y}(t-2), \hat{Y}(t-3), u(t-1), \dots, u(t-3), W) \\ &= F(F(\hat{Y}(t-2), \hat{Y}(t-3), \hat{Y}(t-4), u(t-2), \dots, u(t-4), W), \hat{Y}(t-2), \hat{Y}(t-3), u(t-1), \dots, W) \\ &= F(F(F(\hat{Y}(t-3), \dots, W), \hat{Y}(t-3), \hat{Y}(t-4), u(t-2), \dots, W), F(\hat{Y}(t-3), \dots), \hat{Y}(t-3), u(t-1), \dots, W) \\ &= G(Y(3), Y(2), Y(1), u(t-1), \dots, u(1), W), \end{aligned} \quad (26)$$

for a function  $G$  which is fairly complex. In other words, equation 24 implicitly makes  $\hat{Y}(t)$  a function of  $Y(3), \dots, W$ . In our tests, we picked  $W$  so as to minimize the following measure of error as a function of  $W$ :

$$E_3 = \frac{1}{2} \sum_{i,j} (Y_i(t) - G_i(Y(3), Y(2), Y(1), u(t-1), \dots, u(1), W))^2. \quad (27)$$

In practice, the idea is simply to *chain* the network forward from initial data, as shown in Figure 10.11 (drawn for a different application), and to minimize the total error across the *entire time trajectory*.

In other words, the classic real-time version of parallel identification treats  $\hat{Y}(t-1)$  through  $\hat{Y}(t-3)$  as *constants*, when adjusting  $W$  so as to minimize error at time  $t$ . The off-line version attempts to account for the impact of the weights in *changing* what these inputs would have been. This approach can be applied in a real-time system as well (section 10.4.6), but the tests reported here were all done off-line. The approach can be applied to any forecasting model—not just ANNs—but the minimization problems become much easier when the backpropagation algorithm is used.

The idea of adapting ANNs so as to minimize equation 27 was first proposed by Werbos [34,35], as an extension of his earlier work on the general *nonneural* case [3,36,37]. The ANN version has also been used by Kawato on data from a simulated robot arm, as part of his “cascade” control design

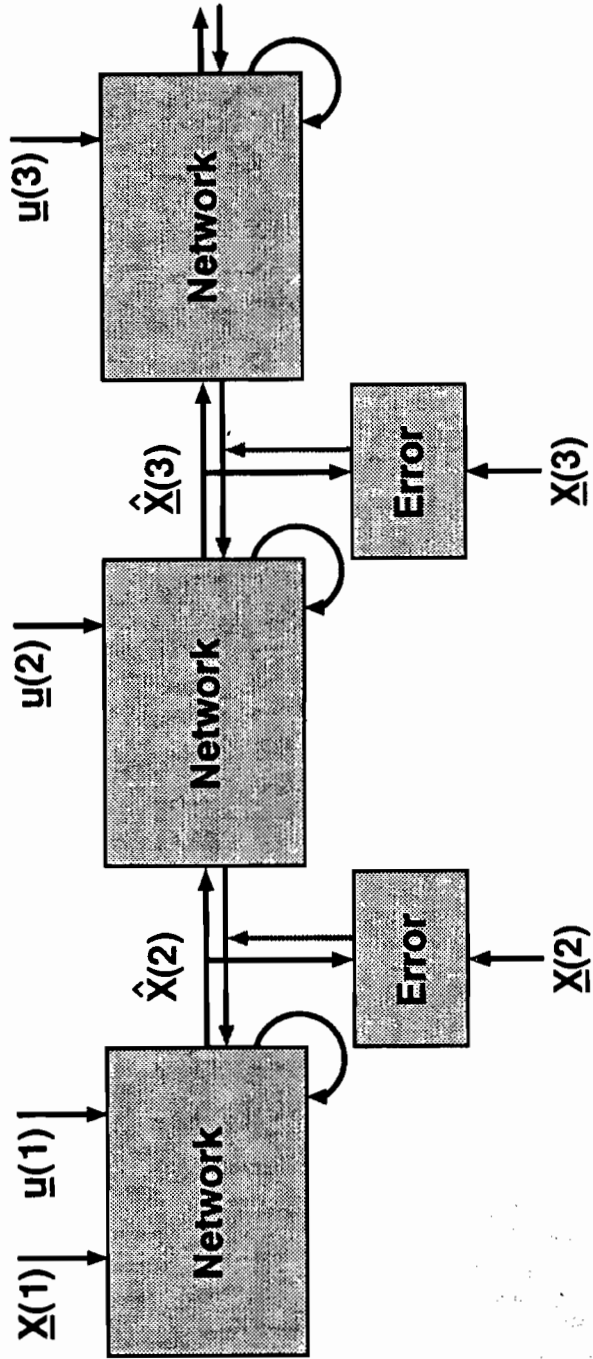


Figure 10.11 The pure robust method.

[38]. Werbos has called this approach “the pure robust method.” In some situations, it is better to use a more complex method, called the compromise method, to be discussed in section 10.4.6. The pure robust method can be rationalized as an alternative maximum likelihood method, based on what control theorists have called an output errors assumption [31–33]; however, section 10.4.6 will argue that this interpretation understates the significance of the method.

In the tests reported here, we compared the conventional maximum likelihood approach (equation 23) against the pure robust method (equation 27) to see which led to better predictions from the ANN. The pure robust method led to multiperiod prediction errors that were smaller by orders of magnitude.

**10.4.1.3. Overview of the Algorithm** It is possible, in theory, to minimize equation 27 by the usual procedure of “backpropagation through time” (BTT). More precisely, we can pick initial values of the weights  $W$  by some sort of random number generator and iterate through the following steps until the weights settle down:

1. Starting from actual data on  $Y$  in times  $t = 1$  through  $t = 3$ , and actual data on  $u$  at all times, use equation 24 to calculate  $\hat{Y}(4)$ ; then use it to calculate  $\hat{Y}(5)$ , and so on up to the end of the training data.
2. Calculate the sum of square error,  $E$ , across all time (based on equation 22).
3. Update all the weights,  $W_i$ , by:

$$\text{new } W_i = \text{old } W_i - \text{learning\_rate} * \frac{\partial E}{\partial W_i},$$

for some small *learning\_rate* determined in ad hoc fashion. The calculation of the derivatives of  $E$  with respect to the  $W_i$  can be done all at once, in one quick sweep through the data, using the backpropagation algorithm.

Unfortunately, the errors  $E$  depend on the weights  $W$  in a highly volatile nonlinear way. There is a kind of Murphy’s law at work here: Highly accurate statistical estimation implies very large second derivatives (because of how one calculates estimation errors in statistics); large second derivatives tend to imply nonsmooth surfaces with steep regions near the optimum; that, in turn, makes it harder to find the minimum of error. In earlier work, Werbos has generally used the following procedure to overcome these problems:

1. Do *not* initialize the weights at random. Instead, initialize them to their maximum likelihood values.
2. Next, adapt the weights using some form of the compromise method, which is similar to the pure robust method, but not so extreme.
3. Then use *those* weights as the initial weights with the pure robust method. (If necessary, repeat step two several times, using stiffer and stiffer forms of compromise.)

This procedure is not guaranteed to find a global minimum. However, it *is* guaranteed to do *better*, in some sense, than the initial weights—the maximum likelihood weights, which are the major alternative we had available to us. In addition, this procedure will work much better if one uses

*adaptive* learning rates, as given in Chapters 3 and 13, or a sophisticated numerical method like one of those discussed by Shanno [24].

For these tests, it was easier and simpler to alternate between a simple version of BTT (as given at the start of this section) and a random search procedure in order to avoid a local minimum. Convergence was still essentially an overnight affair, but this was easier for us than spending weeks to upgrade the computer program. In addition, we used an iterative procedure to decide on the number of hidden units in each network. The remainder of this section will describe the remaining elements of the algorithm in more detail.

**10.4.1.4. Calculation of the Derivatives of Error** The derivatives of  $E_3$ , as defined in equation 27, can be calculated, in principle, by using the *conventional* chain rule for partial derivatives. However, this calculation is very tedious and very subtle. In addition, it is not a “linear” kind of calculation, which can easily be automated. Even the best engineers make errors very easily when they try to calculate partial derivatives in an ad hoc way, through very complex systems.

Werbos [3] developed an alternative chain rule in 1974 for use with ordered dynamical systems, which is more “linear” (i.e., straightforward) and was proven to yield the correct derivatives. Section 10.6 describes the use of this chain rule in more detail.

In our application, it was easiest to write out the forwards equations which were used to calculate error, and then to apply the chain rule for ordered derivatives *directly* to our system, in order to work out the derivative equations. Section 10.8 will provide a more general version of these equations; here, we will report exactly on the version used in these tests [29], developed by Su, for the sake of the intuition that it may provide.

The forwards equations of the MLP were represented as:

$$v^l(t) = W^l x^{l-1}(t) \quad (28)$$

$$x^l(t) = s_l(v^l(t)), \quad l = 1, 2, \dots, L, \quad (29)$$

where  $L$  denotes the output layer (there are  $L + 1$  layers in all), and  $v^l(\cdot)$  and  $x^l(\cdot)$  are the voltage vector and the output vector, respectively, from the  $l$ th layer. The various vectors, matrices, and functions are defined as:

$$W^l \triangleq \begin{bmatrix} 1 & 0 & \dots & 0 \\ W_{1,0}^l & W_{1,1}^l & \dots & W_{1,n}^{l-1} \\ W_{2,0}^l & W_{2,1}^l & \dots & W_{2,n}^{l-1} \\ \vdots & \vdots & \dots & \vdots \\ W_{n,0}^l & W_{n,1}^l & \dots & W_{n,n}^{l-1} \end{bmatrix} \quad (30)$$

$$s_l(v) \triangleq (1, s(v_1), \dots, s(v_n))^T \quad (31)$$

$$v^i(t) \triangleq (1, v_1(t), \dots, v_n(t))^T \quad (32)$$

$$x^i(t) \triangleq (1, x_1(t), \dots, x_n(t))^T. \quad (33)$$

Here,  $W_{j,0}^i$  are bias weights and  $s(\cdot)$  is the sigmoid function defined in equations 10 and 12. In addition, an input neuron can be represented as a neuron with a linear activation function; in other words,  $x^0(t) = v^0(t)$ . These equations, combined with equations 22 and 24, complete our description of how we calculated total error as a function of the weights.

Starting from those system equations, the chain rule for ordered derivatives leads very simply and directly to the following relationships:

$$\frac{\partial^* E}{\partial W_{ji}^*} = \sum_{\tau=1}^T \frac{\partial^* E}{\partial v_j^i(t)} \cdot \frac{\partial' v_j^i(t)}{\partial W_{ji}^*} \quad (34)$$

$$\frac{\partial^* E}{\partial v_j^i(t)} = \frac{\partial^* E}{\partial x_j^i(t)} \cdot \frac{\partial' s_j^i(t)}{\partial v_j^i(t)} \quad (35)$$

$$\frac{\partial^* E}{\partial x_j^i(t)} = \frac{\partial' E}{\partial x_j^i(t)} + \sum_{k=1}^{n^i} \frac{\partial^* E}{\partial v_k^{i+1}(t)} \cdot \frac{\partial' v_k^{i+1}(t)}{\partial x_j^i(t)} \quad (36)$$

and  $\partial/\partial x_j^i(t) = 0$  for all  $i \neq L$ , where  $\partial^*$  indicates ordered derivatives and  $\partial'$  indicates conventional partial derivatives. These conventional partial derivatives are calculated by differentiating the functions for  $E$ ,  $v$ , and  $x$  exactly as they are written in equations 22, 28, and 29 without any substitutions. Let us define:

$$F_{-v_j^i}(t) = \delta_j^i(t) = \frac{\partial^* E}{\partial v_j^i(t)}.$$

With this notation, equation 34 looks very similar to the original delta rule of Widrow and Hoff [39]:

$$\frac{\partial^* E}{\partial W_{ji}^*} = \sum_{\tau=1}^T F_{-v_j^i}(t) x_i^{i-1}(t) = \sum_{\tau=1}^T \delta_j^i(t) x_i^{i-1}(t). \quad (37)$$

In other words, the change of the weights  $W^i$  that connect the  $(i-1)$ th layer to the  $i^{\text{th}}$  layer is determined by the product of the  $(i-1)^{\text{th}}$  layer's output and the  $i^{\text{th}}$  layer's delta. Thus, the key to updating weights in the training phase is to calculate the deltas, the  $F_{-v}$  terms, the derivative feedback to the voltage variables. This calculation can be carried out as follows:

$$F_{-v_j^i}(t) = \frac{\partial^* E}{\partial x_j^i(t)} s_j^i(t) \quad (38)$$

$$\frac{\partial^* E}{\partial x_j^l(t)} = \frac{\partial' E}{\partial x_j^l(t)} + \sum_{k=1}^{n+1} F_{-v_j^{k+1}}(t) W_{kj}^{k+1}. \quad (39)$$

Equation 39 contributes to propagating information about error backwards from the  $(l+1)^{\text{th}}$  to the  $l^{\text{th}}$  layer through the deltas recursively. Note that:

$$s_j^l(t+1) \triangleq \frac{\partial s_j^l(t)}{\partial v_j^l(t)} = x_j^l(t)(1-x_j^l(t)), \quad (40)$$

for layers  $l$  where the sigmoid function is used in equation 29; however, for linear neurons (e.g., input neurons), this  $s'$  term will simply equal one. Nevertheless, equation 39 has a slightly different form, depending on whether we calculate error in the traditional way (equation 23) or in the pure robust way (equation 27).

**Case 1:** When we calculate error in the traditional way, equation 39 becomes exactly the same as in basic backpropagation or the “generalized delta rule” [23], i.e.:

$$F_{-v_j^L}(t) = e_f(t) s_j^L(t) \quad (41)$$

$$F_{-v_j^h}(t) = \left( \sum_{k=1}^{n+1} F_{-v_k^{h+1}}(t) W_{kj}^{h+1} \right) s_j^h(t) \quad (42)$$

where  $e_f(t)$  is defined as  $x_f^L(t) - Y_f(t)$ , and  $h \neq L$  indicates hidden layers.

**Case 2:** When the ANN output  $x^L(t)$  replaces  $Y(t)$  as the input to the network, then changes of weights will affect  $x^L(t+1)$  and thus affect  $x^L(t+2)$  all the way up to  $x^L(t)$ . Therefore, the second term of equation 39 has to account for this chaining from time  $t=0$  to  $t=T$ . As mentioned earlier, the input layer at time  $t+1$  can be considered as a part of the  $(L+1)^{\text{th}}$  layer of the network at time  $t$  (see Figure 10.12). When calculating the delta of the output layer at time  $t$ , one has to calculate the delta for the input neurons at times  $t+1$  up to  $t+k$ , all of which are directly connected to the corresponding output neurons at time  $t$ . (Again, in our tests, we used a  $k$ —a basic time lag—of 1 in some tests, and 3 in others.) It is easy to show that:

$$F_{-v_j^0}(t) = \sum_{k=1}^n F_{-v_k^1}(t) W_{kj}^1 \quad (43)$$

since the input neurons are linear. Therefore, the delta for an output neuron becomes:

$$F_{-v_j^L}(t) = \left( e_f(t) + \sum_{\tau=1}^k F_{-v_k^0}(t+\tau) \right) s_j^L(t). \quad (44)$$

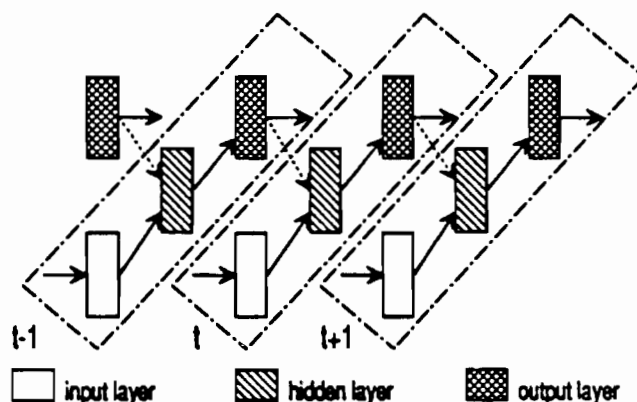


Figure 10.12 Another view of the time-lag recurrent network.

Here,  $*$  indicates the delta of an appropriate neuron to which the output neuron is connected. In equation 44, the delta propagates the required information all the way from  $t = T$  back to the current time  $t$ ; therefore, the delta rule for a time-lagged recurrent system is referred to as the backpropagation through time (BTT) algorithm [22,40]. Likewise, we can easily define a similar rule for training networks where equation 28 is generalized so that  $v'$  depends on  $x(t-1)$  in addition to  $x(t)$ , as in FIR neural networks (see section 10.4.5 and section 10.8).

**10.4.1.5. Random Search Algorithm** Random search methods have been widely used in classical optimization problems. It has been proven that certain random search methods can always reach the global minimum of a nonlinear function. Baba [41] has pointed out that the random search algorithm by Solis and Wets [42] can lead to a global minimum for training a multilayer perceptron. Morris and his co-workers [43] have also utilized a random search algorithm (Chemotaxis algorithm) for neural network training [44]. In these tests, Solis' and Wets' algorithm was used for robust training whenever a convergence problem was encountered. This algorithm, according to its authors, ensures eventual convergence to a global minimum of the objective function. (In actuality, of course, there is no way to *guarantee* global convergence in less than exponential time; see section 10.4.6 for related issues.) The procedure was as follows:

1. Select an initial estimate for  $W^k$  and let  $b^0 = 0$ .
2. Generate a Gaussian random vector  $e^k$  with a predefined variance and a mean of  $b^k$ .
3. If  $E(W^k + e^k) < E(W^k)$ , set  $W^{k+1} = W^k + e^k$  and  $b^{k+1} = 0.2b^k + 0.4e^k$ .
4. If  $E(W^k + e^k) \geq E(W^k) > E(W^k - e^k)$ , set  $W^{k+1} = W^k - e^k$  and  $b^{k+1} = b^k - 0.4e^k$ .
5. Otherwise, let  $W^{k+1} = W^k$  and  $b^{k+1} = 0.5b^k$ .
6. Repeat step two until convergence.



**10.4.1.6. Determining the Number of Hidden Units** Since sufficient hidden neurons are essential for a network to approximate a nonlinear function, the determination of the number of hidden neurons is crucial in the training phase. Many researchers have considered this issue, although no general and efficient method is available. In our tests, we simply used a cross-validation process as a procedure for determining the number of hidden units. First, the number of hidden neurons for the maximum likelihood version of the network is determined. Then, for ease of comparison, the number of hidden units for the robust version is set equal to that of the maximum likelihood version.

The cross-validation procedure was as follows:

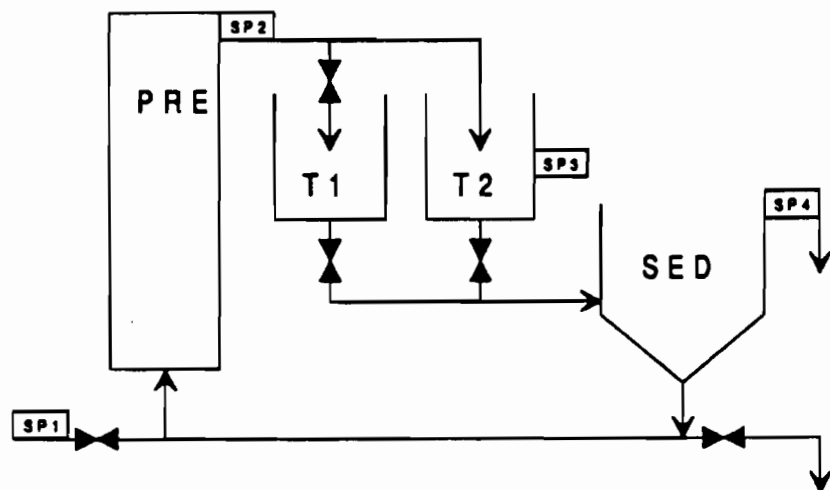
1. Split the data set into a training set and a testing set.
2. Start with a small number of hidden units.
3. Initialize all the weights to zero as discussed below.
4. Apply the generalized delta rule to update the weights.
5. Validate the updated model with the testing set.
6. Repeat the training process until the prediction error of the testing set reaches a minimum.
7. Add an additional hidden neuron to the network.
8. Repeat step three until adding additional hidden neurons increases the prediction error on the testing set.

In order to have a common starting point for different-size networks, all the weights are initialized to zero when determining the number of hidden neurons. A network with all weights equal to zero, regardless of the number of hidden neurons, will have the property that the prediction errors for both training and testing sets are proportional to the sampling variance of the whole data set. However, for a network with all weights equal to zero, the final results for the weight matrix will result in all the hidden neurons being identical to each other. To avoid this problem, different learning rates are assigned to the weights connected with different hidden neurons. In this work, the learning rate is multiplied by a linearly decreasing factor varying with the hidden unit so that the learning rate is actually a vector, equal to a global learning rate multiplied by  $(1, 1, \dots, 2/n', 1/n')$ . Note that the bias weight has the same learning rate as the first hidden neuron. From an optimization point of view, one can consider that the gradient is being multiplied by the matrix  $\text{diag}[1, 1, \dots, 2/n', 1/n']$ , which is positive-definite and invertible. It is easy to prove that an optimization algorithm using such a transformed gradient can lead to a minimum in the error.

## 10.4.2. Example Number One: A Wastewater Treatment System

**10.4.2.1. Description of the System and the Network** The data for this study were provided by Pedersen, Kummel, and Soeberg [45], to whom we owe many thanks.

A wastewater treatment plant is a very complex process, especially when biological removal of nitrogen, phosphorous, and organic material (N, P, and C) is implemented, as in the research of Pedersen et al. [45]. The pilot plant, as sketched in Figure 10.13, is fed with clarified municipal wastewater, in which the large particles, sand, and oils have been removed. The process is carried out in four vessels, in which the biological removal of phosphorous, nitrogen, and organic material takes place. The throughput of the pilot plant is on the order of 60 liters per hour. The approximate



**Figure 10.13** The process flow diagram of the wastewater treatment plant.

holding volumes of the pretreatment column (PRE), the two aeration tanks (T1 and T2), and the sedimentation tank (SED) are 200, 800, 800, and 1,100 liters, respectively.

The feed to the pretreatment column is a mixture of raw wastewater and return sludge. The pretreatment column is maintained anaerobic, and the main biological process is carried out by the phosphorus accumulating bacteria. The two aeration tanks T1 and T2 are operated in an alternating manner that is scheduled prior to the operation. The nitrification and denitrification processes take place in these two tanks alternatively. In the sedimentation vessel, the heavier sludge falls out of suspension. From the top of the vessel, the process effluent is removed from the system. The sludge is returned via the sludge return pump and is mixed with the fluid entering the pretreatment column. The sludge concentration is held approximately constant through intermittent removal of sludge from the system at the point where it exits the sedimentation vessel. Samples are taken from four different locations as indicated in Figure 10.13, in which SP<sub>1</sub> through SP<sub>4</sub> measure the concentrations of  $\text{NH}_4^+$ ,  $\text{NO}_3^{2-}$  and  $\text{PO}_4^{3-}$  in the incoming stream, the pretreatment column, the aeration tanks, and the sedimentation vessel, respectively. The oxygen consumption rates at the aeration tanks are also recorded. A time series of four days' data is used to model the pilot plant. The data set is obtained by sampling the system every seven minutes. This sampling results in nine hundred data points in total. All the variables are linearly scaled between 0.1 to 0.9. The ranges of all the variables are given in Table 10.2. The data is split into a training set (the first six hundred samples) and a testing set (the remaining three hundred samples). The outputs of interest are the concentration measurements for the three processing units (nine variables). The inputs are the concentrations of the incoming stream and the oxygen consumption rates (five variables).

One can construct a network to model the whole plant as a 5 by 9 multi-input–multioutput (MIMO) system. Alternatively, in this study we used three smaller networks to model the three processing

**Table 10.2** The range for each of the measurements for the wastewater plant

unit: mg/l	Inlet	PRE	T12	SED
$[NH_4^+]$	.0366 49.51	4.956 28.934	.0366 11.768	.0061 1.147
$[NO_3^-]$	0.000 11.58	.000 .824	0.0214 16.290	.4364 12.50
$[PO_4^{3-}]$	.1953 8.287	11.130 48.276	0.5066 21.658	.6195 9.338

**Table 10.3** The number of neurons for each layer of the neural network for the wastewater plant and the catalytic reformer

	PRE	T12	SED	REF
input	6	8	18	21
hidden	5	5	6	6
output	3	3	3	2

units individually. The three processing units are the pretreatment column (PRE), the aeration tanks (T12), and the sedimentation vessel. The pretreatment column is a 3 by 3 system, the aeration tanks, 5 by 3, and the sedimentation vessel, 3 by 3. A first-order dynamic model ( $k = 1$ ) is chosen for PRE and T12, and a third order for SED. The order of the dynamic model determines the number of input neurons needed for the networks. The size of the network for each subsystem is summarized in Table 10.3.

**10.4.2.2. Results on the Wastewater Plant** After the ANN is trained by the maximum likelihood procedure, the one-step-ahead prediction by the ANN for each subsystem appears to be exceptionally good. These results for the three subsystems (PRE, T12, and SED) are shown in Figures 10.14, 10.15, and 10.16, respectively. In principle, one can use these models for nonlinear model predictive control (MPC), as described in section 10.3. In MPC, the model usually predicts many time-steps ahead into the future. Therefore, the feedforward network, which predicts one step ahead, has to be iterated; in other words, the network has to be chained to itself to go as far as needed into the future. For example, if a predicted output in the future, like  $\hat{Y}(t+k)$ , is needed at any time  $t$ , then the predicted output  $\hat{Y}(t+1)$  has to replace the actual output  $Y(t+1)$  measured from the system, and  $\hat{Y}(t+2)$  must replace  $Y(t+2)$  and so on, all the way up to  $t+k-1$ , because the actual system outputs in the future are not yet known. Therefore, the errors of importance to MPC are the multiperiod forecasting errors, the kinds of errors minimized in the pure robust method. In this situation, the gap between  $Y$  and  $\hat{Y}$  can grow in a cumulative fashion over time, leading to large errors.

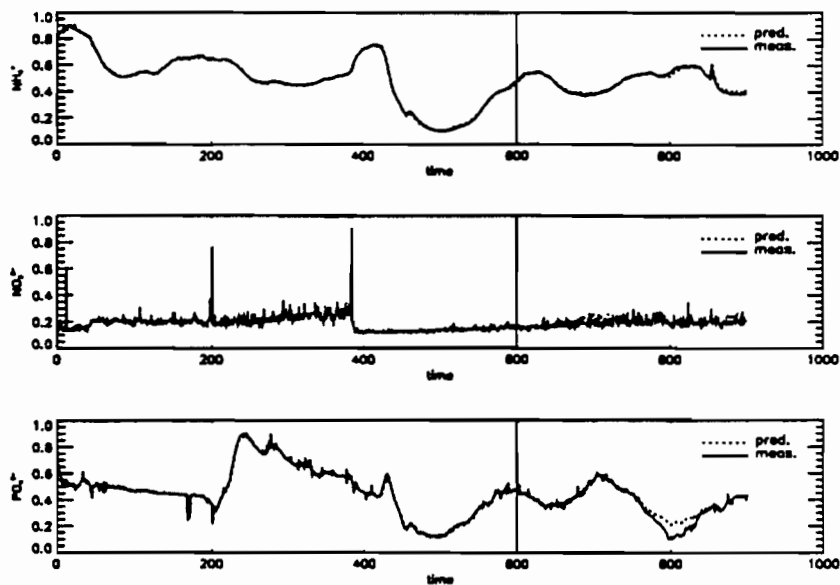


Figure 10.14 FFN: one-step ahead prediction for PRE.

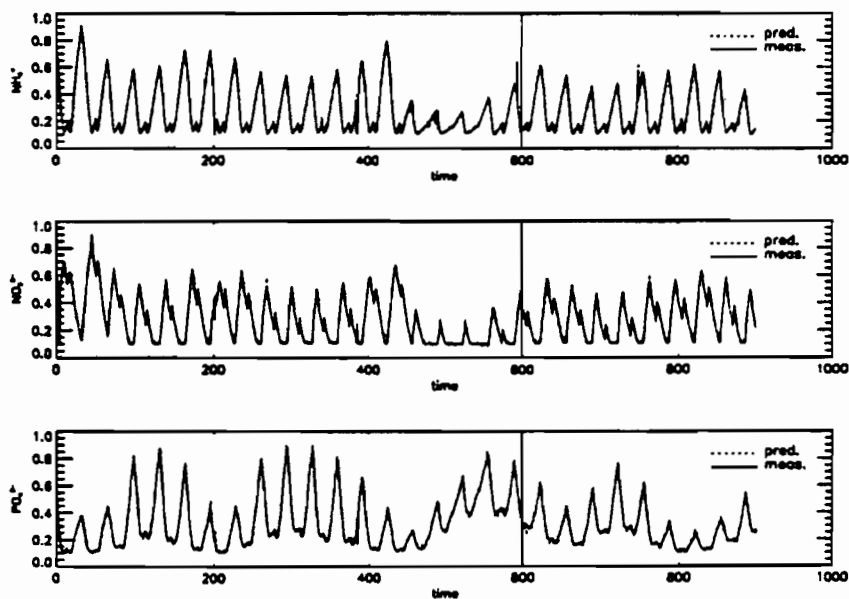


Figure 10.15 FFN: one-step ahead prediction for T12.

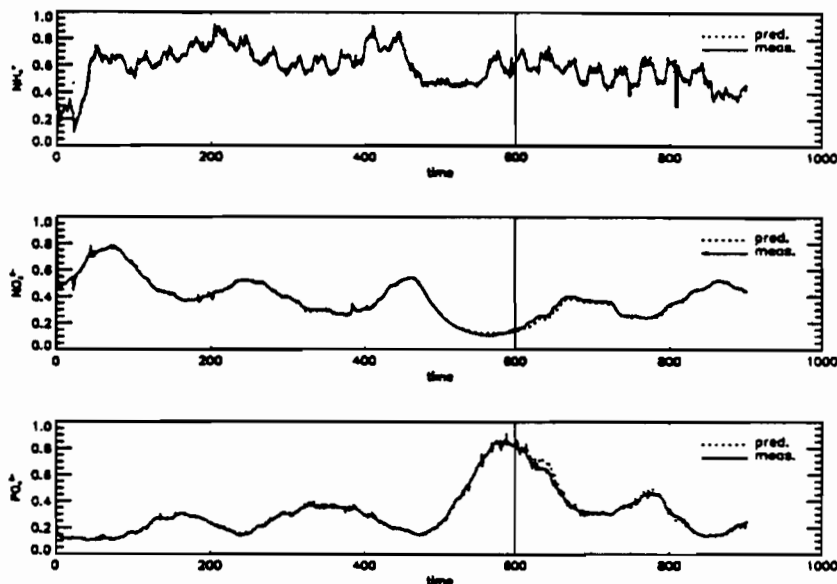


Figure 10.16 FFN: one-step ahead prediction for SED.

As an extreme example, the maximum likelihood net (MLN) is chained to predict the future system output all the way from  $t = 1$  to  $t = 900$ , including the training set as well as the testing set. As shown in Figures 10.17 through 10.19, the MLNs do not give consistently good results for the long-term predictions on the three subsystems as they do for the one-step-ahead predictions. However, the MLN, a one-step-ahead predictor, can give a fairly good long-term prediction for the T12 subsystem. For the PRE and SED subsystems, as shown in Figures 10.17 and 10.19, the MLNs do not give reasonable long-term predictions at all. The reason for such inconsistent results from the use of MLNs is simply that they are not trained to make long-term predictions.

On the other hand, the networks adapted by the pure robust method are trained to predict the whole trajectory of the system output all the way from  $t = 1$  to  $t = T$  ( $T = 600$ ). Figures 10.20 through 10.22 show the results from the pure robust networks (PRNs) in predicting from time  $t = 1$  to  $t = 900$  (105 hours ahead). Clearly, the PRN is significantly superior to the MLN for long-term predictions, especially for the PRE and SED subsystems. For comparison, the root mean square (RMS) of the prediction errors for each subsystem are given in Table 10.4. Note that the prediction errors are based on the scaled variables.

### 10.4.3. Example Number Two: A Catalytic Reforming System

**10.4.3.1. Description of the System and the Network** The data for this test came from Mobil Corporation, whose help is gratefully acknowledged.

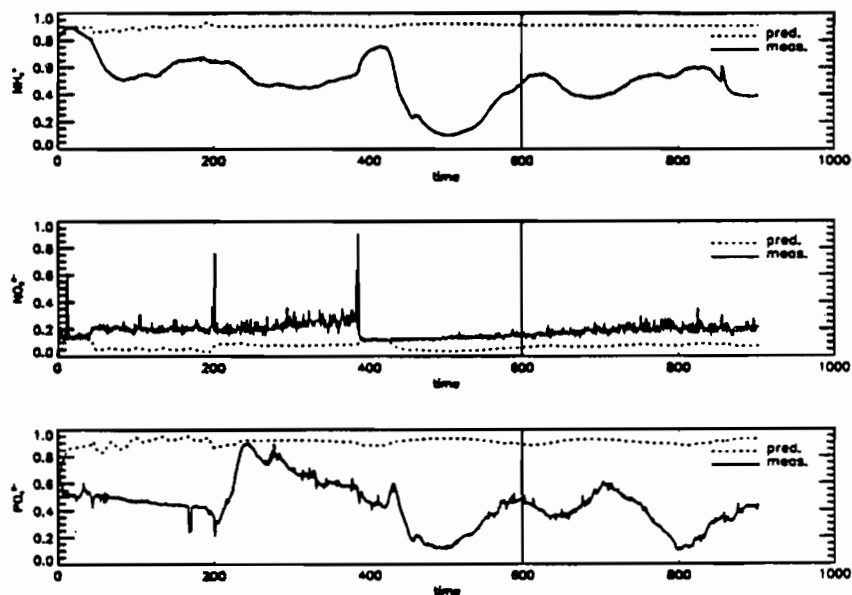


Figure 10.17 FFN: long-term prediction for T12.

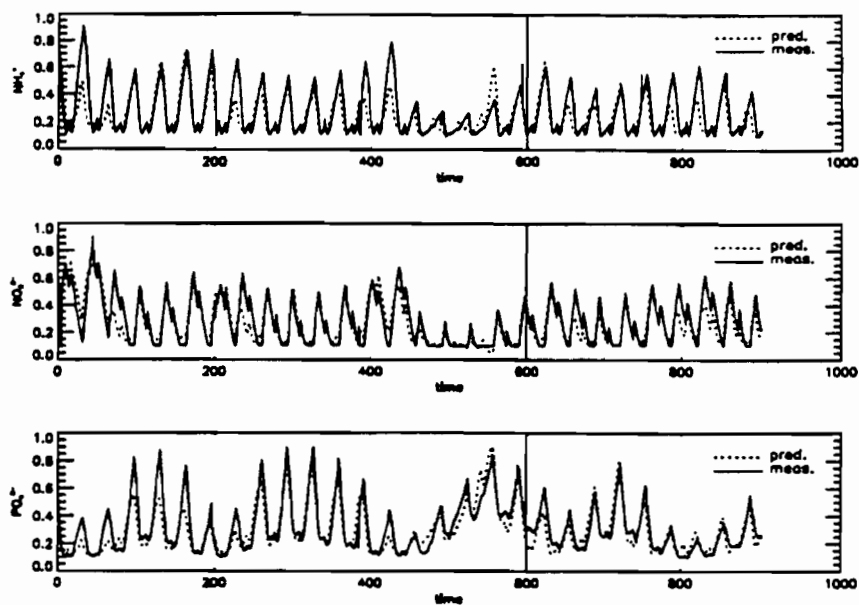


Figure 10.18 FFN: long-term prediction for T12.

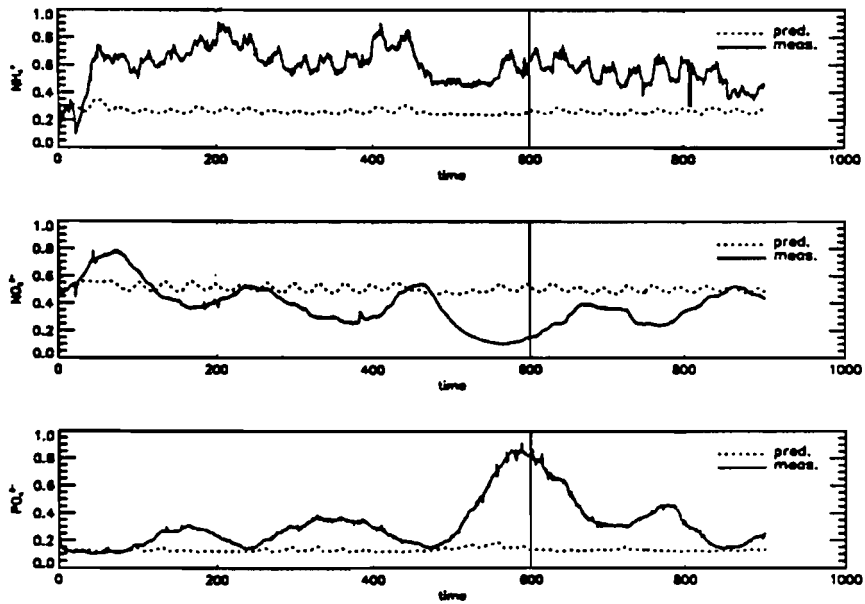


Figure 10.19 FFN: long-term prediction for SED.

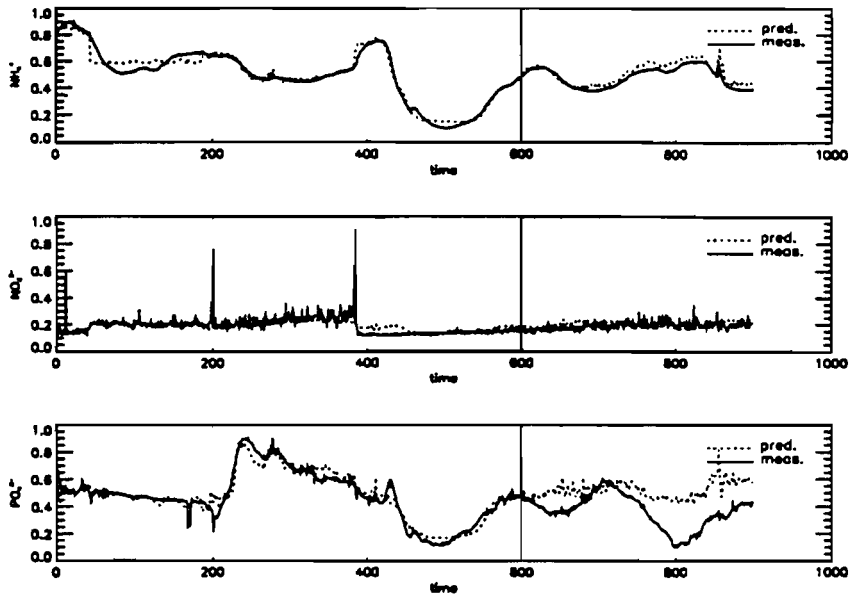


Figure 10.20 REC: long-term prediction for PRE.

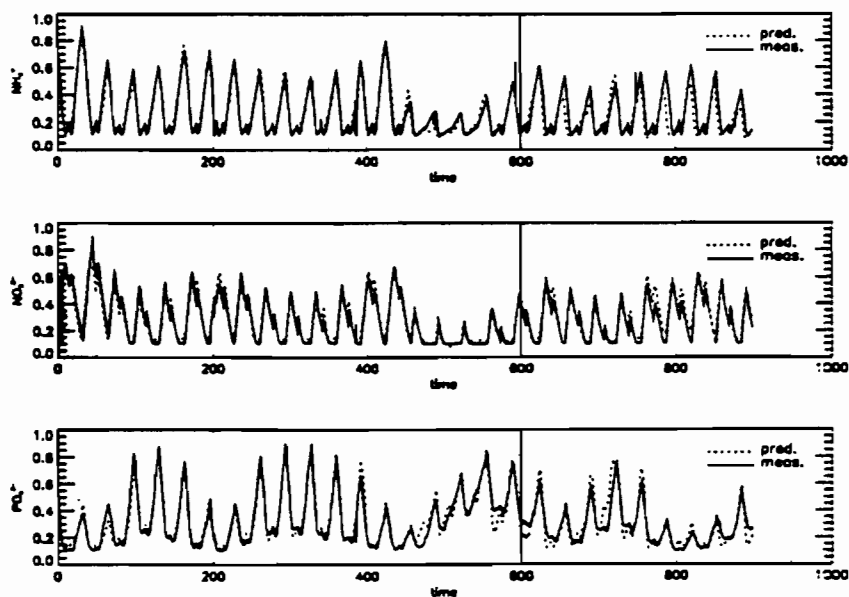


Figure 10.21 REC: long-term prediction for T12.

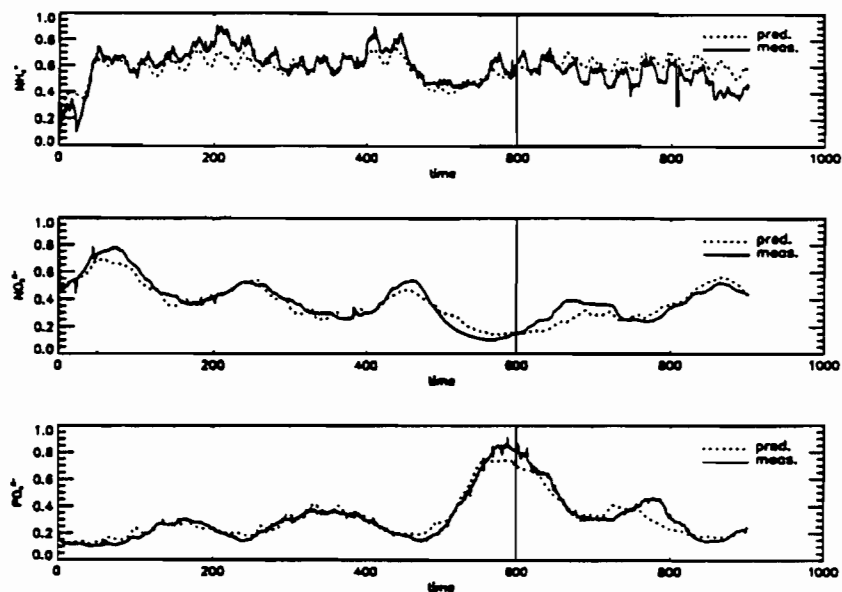


Figure 10.22 REC: long-term prediction for SED.



Since the 1970s, the catalytic reformer has become one of the key systems in oil refineries, and catalytic reformat has furnished a relatively high portion of the gasoline used in the United States [46]. In catalytic reforming, the structure of the hydrocarbon molecules is rearranged to form higher octane aromatics that can be used to increase the octane number of gasoline. The catalytic reformer carries out four major reactions: (1) dehydrogenation of naphthenes to aromatics; (2) dehydrocyclization of paraffins to aromatics; (3) isomerization; and (4) hydrocracking. Catalyst activity is reduced during operation by coke deposition and chloride loss, and it can be restored by chlorination following high temperature oxidation of the carbon.

A typical catalytic reforming process consists of a series of reactors and heaters, and a fractionator, as shown in Figure 10.23. The typical feed to a catalytic reformer is the heavy straight-run (HSR) gasoline and naphtha. Material that will deactivate the catalyst is removed prior to the process. The pretreated feed and recycle hydrogen are heated to about 900° F before entering the first reactor. In the reactors, the reactions will cause temperature drops. The effluent from a reactor has to be reheated to maintain the reaction rate before it enters the next reactor. As the flow goes down the reactors, the reaction rate decreases and the reactors become larger, and the reheat needed becomes less. Usually three reactors are enough to provide the desired degree of reaction. The effluent from the last reactor is cooled and separated into hydrogen-rich gaseous and liquid streams. The gaseous stream is recycled to the first reactor and the liquid stream is sent to the fractionator to be debutanized. The reformat is drawn out from the fractionator.

The data for the catalytic reforming system (REF) contains five inputs (denoted by  $u$ 's) and two outputs (denoted by  $Y$ 's). Due to a secrecy agreement, the detailed description of the measurements and the range of each variable are not given. As with the wastewater plant, the variables taken from the reformer are linearly scaled to between 0.1 and 0.9. The data contains 392 data points in total, of

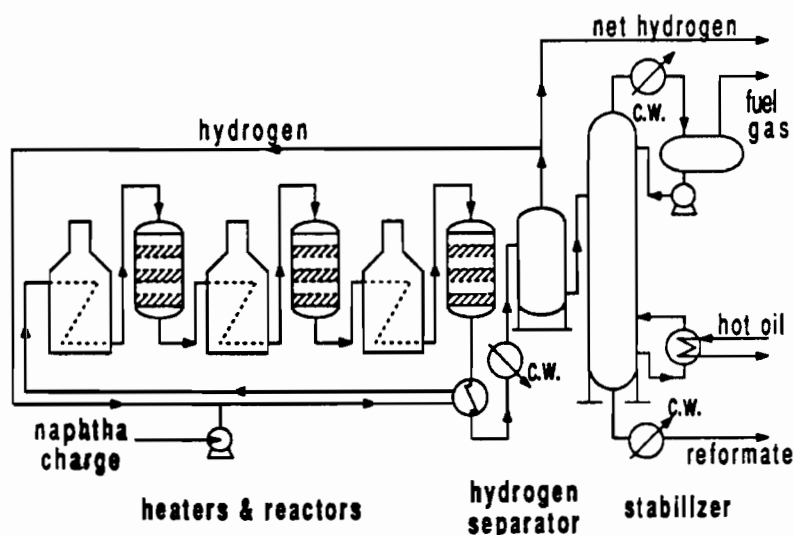


Figure 10.23 REC: long-term prediction for PRE.

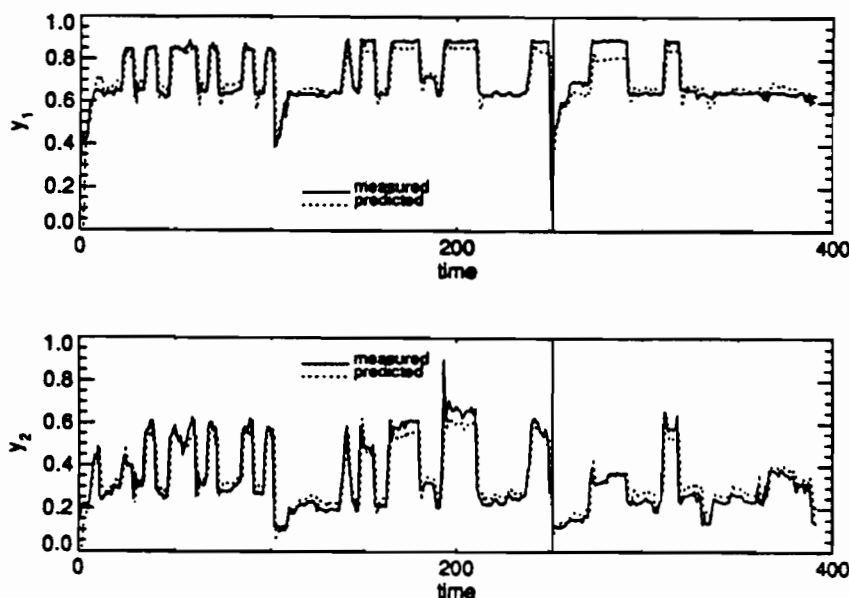


Figure 10.24 FFN: one-step ahead prediction for REF.

which the first 252 points are used for training and the remaining 140 for testing. In this case, a third-order 5 by 2 model is used. The size of the network used here is described in Table 10.3.

**10.4.3.2. Results on the Catalytic Reformer** In the case of one-step-ahead prediction, the network trained by the conventional maximum likelihood approach predicts the testing set of the catalytic reformer as well as it did with the wastewater system (Figure 10.24). However, it did not perform as well in tests of multiperiod predictions. In the most extreme test, the network was chained to make long-term predictions from  $t = 1$  to  $t = 392$  (including both training and testing sets). As in the wastewater example, the maximum likelihood network (MLN) performed much worse on these tests than it did for one-step-ahead prediction (Figure 10.25). Strictly speaking, one should not use an MLN for multiple-step predictions since it is trained to make single-step predictions. The multiple-step prediction requires iteration or chaining, and conventional maximum likelihood training does not take such chaining into account. On the other hand, the pure robust (PRN) version of the network was trained to make long-term predictions. After training, it gives much better results than the MLN for long-term predictions (Figure 10.26). For comparison, the RMS of the prediction errors for the reformer are given in Table 10.4. Notice that the prediction errors are calculated based on the scaled variables.

#### 10.4.4. Overall Evaluation of Performance and Implications

As described earlier, networks trained by conventional maximum likelihood procedures and networks trained by the pure robust method were used to make long-term predictions. The number of prediction

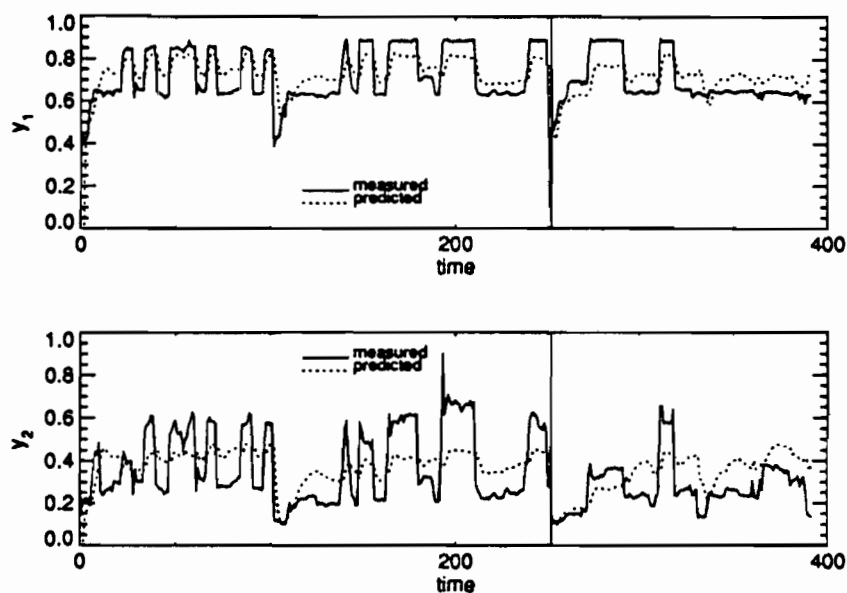


Figure 10.25 REC: long-term prediction for REF.

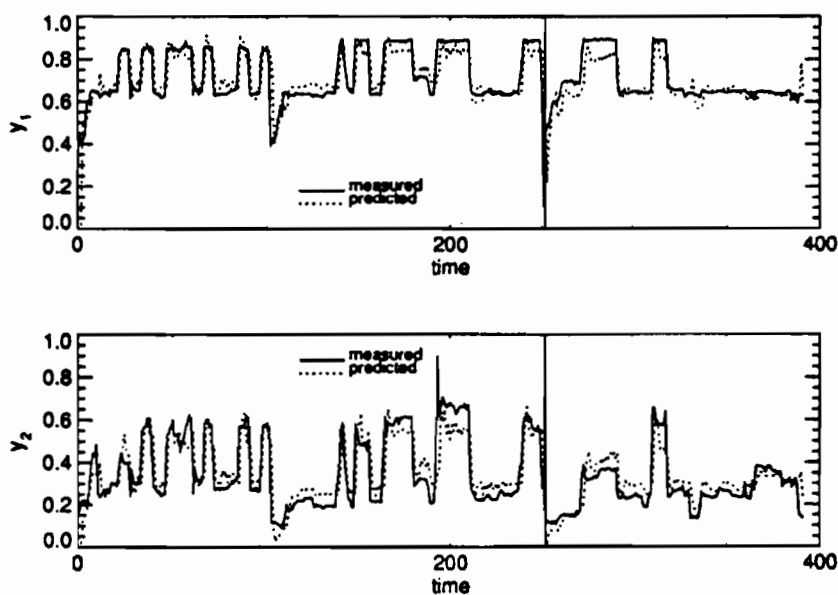


Figure 10.26 REC: long-term prediction for REF.

**Table 10.4** RMS prediction errors for the wastewater plant and the catalytic reformer

(a) the wastewater plant				
unit: 1/100		PRE	T12	SED
<i>one-step ahead</i> FFN training	$NH_4^+$	3.5	4.3	2.8
	$NO_3^{2-}$	5.2	3.8	3.9
	$PO_4^{3-}$	3.6	3.2	1.7
testing	$NH_4^+$	1.5	3.1	3.8
	$NO_3^{2-}$	4.1	2.4	1.7
	$PO_4^{3-}$	4.0	1.8	2.8
<i>long-term</i> FFN training	$NH_4^+$	45.3	12.5	31.3
	$NO_3^{2-}$	13.6	8.5	21.7
	$PO_4^{3-}$	47.0	10.0	18.7
testing	$NH_4^+$	43.1	10.2	23.6
	$NO_3^{2-}$	12.0	8.6	11.8
	$PO_4^{3-}$	54.8	7.4	18.5
<i>long-term</i> REC training	$NH_4^+$	4.0	4.5	8.1
	$NO_3^{2-}$	5.7	5.6	7.0
	$PO_4^{3-}$	5.1	5.3	4.2
testing	$NH_4^+$	4.0	9.5	11.4
	$NO_3^{2-}$	3.8	6.0	6.2
	$PO_4^{3-}$	18.8	7.0	7.0
(b) the catalytic reformer				
unit: 1/100	training		testing	
	y1	y2	y1	y2
FFN (1-step ahead)	8.2	9.1	5.5	6.2
FFN (long-term)	10.4	14.5	8.6	12.5
REC (long-term)	8.6	9.9	5.6	6.3

steps (i.e., prediction horizon) was equal to the total number of samples: 900 for the wastewater system and 392 for the catalytic reformer. However, if the networks were used for model predictive control, the prediction horizon would be much smaller, e.g., twenty or thirty steps ahead into the future. It is important to know how well both networks perform within a smaller prediction horizon. In this example, the width of the prediction horizon,  $p$ , is varied from 1 to 30, and the prediction errors are calculated for  $p = 1, 5, 10, 15, 20, 25$ , and 30. The RMS prediction errors are plotted against  $p$  in Figure 10.27. Notice that the prediction error is calculated only at the end of the prediction horizon.

The RMS errors are calculated for all data points in the testing set: 300 for the wastewater system and 140 for the reformer. In order to predict the first  $p$  points of the testing set, training data that immediately precede the testing data were used for predictions. As shown in the figure, the prediction errors of a maximum likelihood network increase significantly as the prediction horizon becomes large, whereas those of a pure robust network are more stable. Although in some cases a maximum likelihood network gives results similar to a pure robust network (e.g., Figure 10.27f), the pure robust network gives consistently better results than a maximum likelihood network in cases where  $p > 5$ . The reason is that conventional maximum likelihood training is based on minimizing errors in one-step-ahead predictions. On the other hand, pure robust training minimizes multiple-step prediction errors.

Even though the pure robust network is normally trained over the whole trajectory of the training data, it can be trained over a smaller trajectory [47]. For example, if the prediction horizon in MPC contains  $p$  time steps, we can train the net over a trajectory of  $p$  time-steps [48]. In this case, one can set up a moving window of width  $p$  along the whole training set. The network is then trained over the trajectory within this moving window. This approach results in a case of semibatchwise backpropagation through time. A similar effect may be obtained at lower cost by using the compromise method, to be discussed in section 10.4.6.

In summary, robust training is more appropriate than conventional training for ANNs to be used in model predictive control.

### 10.4.5. Alternatives to the MLP in Control Theory and Neural Nets

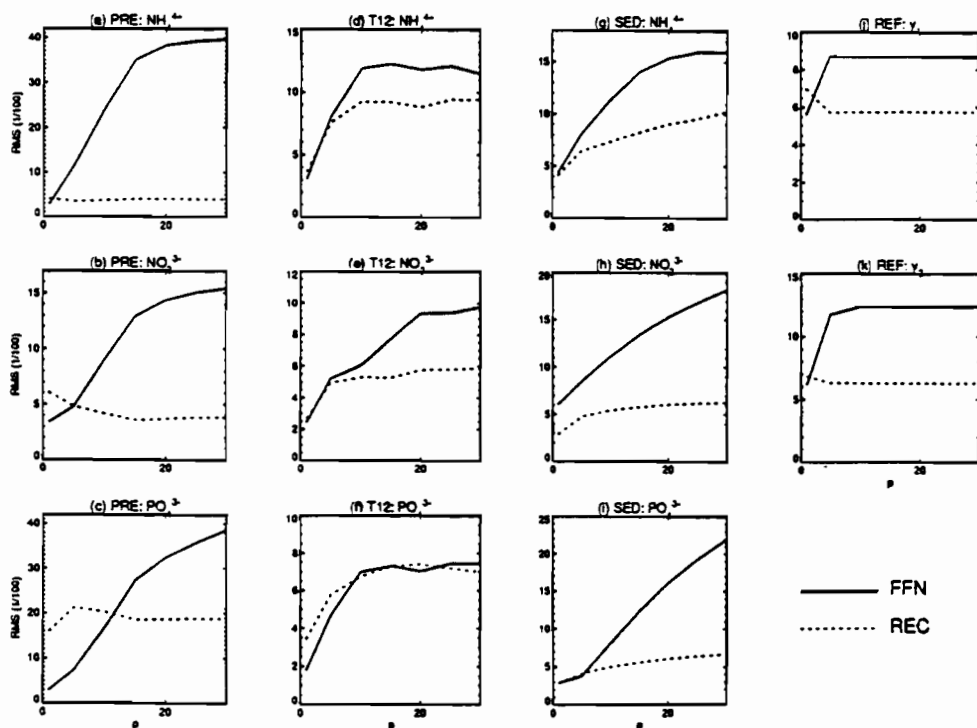
**10.4.5.1. Summary** The empirical results of sections 10.3 and 10.4 were all based on the multilayer perceptron (MLP)—a very simple form of ANN. Many control theorists have argued that MLPs are essentially static models, and that true system identification requires more dynamic models analogous to those used in control theory. This argument is correct in principle; however, there are alternative forms of ANN that provide nonlinear generalizations of the usual dynamic models.

In practical applications, simple ANNs are often good enough. In some applications, simple MLPs cannot forecast  $Y(t+1)$  well enough, but the problem can be solved by a simple change of variables (e.g., forecasting  $Y(t+1) - Y(t)$  rather than  $Y(t+1)$  [49]). This subsection will discuss alternative models and functional forms for use in applications where MLPs are *not* good enough. When an MLP model appears adequate at first, but leads to limited capabilities in control applications, then it is important to reexamine that model (as we did in section 10.3), and to consider using an alternative model or adaptation procedure.

**10.4.5.2. Classical NARMAX and Kalman Models** Among the models used in control theory and statistics, one of the better known is the classical NARX (Nonlinear AutoRegressive model with exogenous variables) model, which may be written:

$$\hat{Y}(t) = F(Y(t-1), \dots, Y(t-k), u(t-1), \dots, u(t-k), W), \quad (45)$$

where  $k$  is the order of the model and  $W$  is the set of model parameters.



**Figure 10.27** Prediction errors (RMS) vs.  $p$  for a FFN and a REC. Notice that there are 300, 300, 300, and 140 data points in the testing set of PRE, T12, SED, and REF systems, respectively. The RMS errors are calculated based on the scaled variables in all cases.

In order to represent a general NARX model, we need to use a class of models that can represent any arbitrary function  $F$ ; since MLPs can approximate any well-behaved vector-valued function  $F$ , they can represent a general NARX model if their inputs and outputs are arranged as in equation 45. The resemblance between equation 45 and equation 20 should make it clear that MLPs can do justice to this class of model. (Strictly speaking, however, there are many nonlinear functions that can be represented more *parsimoniously* by adding a feature called “simultaneous recurrence,” which is discussed in section 10.4.6.4. Simultaneous recurrence is one more technique for improving performance at the cost of doing more work, a technique that can be used to upgrade any of the models to be discussed in this section.)

In their classic introductory textbook [50], Box and Jenkins discuss a variety of *stochastic* process models, including the AR( $p$ ) model, based on a special case of equation 45:

$$Y(t) = w_1 Y(t-1) \dots + w_p Y(t-p) + e(t), \quad (46)$$

where  $e(t)$  is a random number with a Gaussian distribution. They also discuss the ARMA(p, q) model, which may be written:

$$Y(t) = w_1 Y(t-1) \dots + w_p Y(t-p) + e(t) + \theta_1 e(t-1) + \dots + \theta_q e(t-q). \quad (47)$$

In theory, an ARMA model of finite p and q can always be represented by an equivalent AR model of infinite order. Superficially, this suggests that AR models (and their nonlinear generalizations) are always good enough to model any process. However, the need for an infinite number of parameters is of enormous practical importance. In any form of identification or estimation, the speed of learning and the quality of generalization (i.e., the estimation errors after a finite database of experience) depend on the number of parameters to be estimated; it is important to reduce the number of parameters as much as possible. Therefore, when there is a possibility that the system to be identified is an ARMA process, it is important to use an ARMA specification or the equivalent.

Box and Jenkins [50] explain how an ARMA (1, 1) process results very simply from an AR (1), process observed through measurement noise, as in:

$$X(t) = wX(t-1) + e_1(t) \quad (48)$$

$$Y(t) = X(t) + e_2(t), \quad (49)$$

where  $e_1$  represents random noise in the system to be controlled, and  $e_2$  represents measurement error in observing that process. If the variable  $Y$  is observed but the variable  $X$  is not, then the behavior of  $Y$  over time fits an ARMA (1, 1) model (albeit with different parameters). The general state-space model of linear control theory is simply a multivariate generalization of equations 48 and 49; therefore, it leads to a vector ARMA process. This is the reason why models of the ARMA type are of great importance in control applications. In order to deal with nonlinear problems, authors like Leontaritis and Billings [51] have recently advocated the use of NARMAX models, which may be written as:

$$\hat{Y}(t) = F(Y(t-1), \dots, Y(t-p), u(t-1), \dots, u(t-p), e(t-1), \dots, e(t-q)), W), \quad (50)$$

where:

$$e(t) = Y(t) - \hat{Y}(t) \quad (51)$$

is a random vector.

An alternative, equivalent way to cope with measurement noise in the linear case is to use Kalman filtering [52,53]. Instead of just modeling the observed vector,  $Y$ , one may try to estimate the underlying state-space vector, which we will denote as  $R$ . (In control theory, the state-space vector is usually denoted as  $x$ , but there are advantages to using a different notation with ANNs.) We arrive at a model that may be written:

$$\hat{Y}(t) = H\hat{R}(t)$$

$$\hat{\mathbf{R}}(t) = \mathbf{A}\mathbf{R}(t-1) + \mathbf{B}u(t) \quad (52)$$

$$\mathbf{R}(t) = \hat{\mathbf{R}}(t) + \mathbf{M}(\mathbf{Y}(t) - \hat{\mathbf{Y}}(t)),$$

where  $\hat{\mathbf{Y}}$  and  $\hat{\mathbf{R}}$  are forecasts based on prior data, and  $\mathbf{R}$  is our best guess as to the true value of  $\mathbf{R}$  at time  $t$ , after the measurements from time  $t$  are available.  $\mathbf{H}$ ,  $\mathbf{A}$ ,  $\mathbf{B}$ , and  $\mathbf{M}$  are matrices of parameters of the model. One consistent way to estimate the parameters in this model is simply to minimize the square error in predicting  $\mathbf{Y}(t)$ . (Strictly speaking, this model does contain redundant parameters, which makes the error-minimizing values nonunique; however, this is not a problem unless there is some other reason to want to estimate specific hidden variables. For example, if one of the unobserved variables represents temperature, and high temperatures cause melting behavior outside of what the linear model predicts, one then needs to use more than just empirical data on the observed variables from normal operating regimes.) There are nonlinear extensions of Kalman filtering, but many researchers consider them cumbersome and less than completely satisfactory.

**10.4.5.3. Time-lagged Recurrent Networks (TLRNs)** The simple ANN models of sections 10.3 and 10.4.1—which are a special case of the NARX model—cannot effectively subsume the NARMAX or Kalman models above. However, there is a class of ANN models, developed by Werbos [21,34,54], which *can* subsume those models: the time-lagged recurrent network (TLRN). In effect, the TLRN provides an efficient nonlinear generalization of the Kalman filter, *when* our goal is to estimate the state vector or to predict  $\mathbf{Y}(t)$  over time. *Unlike* the classical ARMA models, the TLRN does not provide a general scheme for describing the *probability distribution* for *alternative* possible values of  $\mathbf{Y}(t)$ ; to make that generalization, see the discussion of the Stochastic Encoder/Decoder/Predictor in Chapters 3 and 13.

In the neural network literature, a “recurrent network” refers to any ANN in which a neuron can use its own output as one of its inputs, or, more generally, in which the flow from inputs to outputs may contain loops. There are two fundamentally different *kinds* of recurrence [54], each of which has its uses: (1) *simultaneous* recurrence, discussed in Chapters 3 and 13; and (2) time-lagged recurrence. In time-lagged recurrence, we simply allow any neuron at time  $t$  to receive, as input, the output of *any* neuron at time  $t-1$ . It is trivial to allow inputs from times  $t-2$  through to  $t-k$  as well [22], but this does not really add any generality in practice; lags greater than 1 can be represented quite effectively by bucket brigades or exponential decays within a lag-1 structure.

The most classical form of TLRN is a multilayer perceptron, MLP, *modified* to allow crosstime connections. The equations are:

$$x_i(t) = X_i(t) \quad 1 \leq i \leq m \quad (53)$$

$$v_i(t) = \sum_{j=0}^{i-1} W_{ij}x_j(t) + \sum_{j=1}^{N+n} W_{ij}'x_j(t-1) \quad m+1 \leq i \leq N+n \quad (54)$$

$$x_i(t) = s(v_i(t)) \quad m+1 \leq i \leq N+n \quad (55)$$

$$\hat{\mathbf{Y}}_i(t) = x_{i+n}(t) \quad 1 \leq i \leq n \quad (56)$$



where  $x_0$  represents the constant 1, where  $N+n$  is the total number of neurons, where  $\mathbf{X}$  is the vector of *all* inputs at time  $t$  (usually made up of  $\mathbf{Y}(t-1)$  and  $\mathbf{u}(t)$ ), and where  $m$  is the total number of input neurons. In practice, it works best to delete (or “zero out” or “prune” most of the weights  $\mathbf{W}$  and  $\mathbf{W}'$ , but equation 54 indicates the full range of connections that are allowed.

In this arrangement, it should be possible to represent the estimated state vector,  $\mathbf{R}$ , through hidden nodes in the network. Any set of filtering equations could be approximated as well as desired by the subnetwork that leads up to the calculation of those nodes. One can adapt this entire system by minimizing square error, exactly as in basic backpropagation. It is straightforward to calculate the relevant derivatives (either for maximum likelihood estimation or for the pure robust method) through the use of backpropagation through time; see Section 10.8.

**10.4.5.4. Sticky Neurons and Their Capabilities** Time-lagged recurrence can yield two additional capabilities, which can be very important in intelligent control:

- “Short-term memory”—the ability to account for phenomena observed many time periods ago, which are not currently observable but are still currently relevant. For example, the person who turns away from a loose tiger—to run—needs to account for the presence of a tiger behind his back even though he can’t see the tiger any more. Likewise, in manufacturing, short-term memory allows you to account for the properties of the last part you made even as you adapt to making a new part.
- The ability to estimate slowly varying process parameters. The need for this is one of the real motives behind classical adaptive control.

In both cases, recurrent hidden nodes can provide the required memory or parameter estimates, in principle. However, the specification in equation 55 does not provide enough stability to make this realistic. An alternative specification [35], inspired by discussions of the Purkinje cells of the cerebellum, is the “sticky neuron” defined by:

$$v_i(t) = \sum_{j=0}^{i-1} W_{ij}x_j(t) + \sum_{j=1}^{N+n} W'_{ij}v_j(t-1). \quad (57)$$

When  $W'_{ij} = \delta_{ij}$ , this system becomes extremely stable, stiff, or sticky. Because it is even stiffer than the usual TLRN, the user is advised—once again—to first adapt a less stiff or less general version of the network in order to arrive at good initial values for the weights. In some cases, one might initially fix the  $W'_{ij}$  to  $\delta_{ij}$  or to  $0.9\delta_{ij}$ , and then adapt these sticky weights only after the other weights have settled down. One might use the gradient-based learning rate adjustment scheme, discussed in Chapters 3 and 13, to adapt a different learning rate for the sticky weights. For system identification applications, one might even consider a more extreme form of sticky neuron, also suggested by Werbos [35]:

$$v_i(t) = v_i(t-1) + \sum W_{ij}(Y_j(t-1) - \hat{Y}_j(t-1)). \quad (58)$$

As with other forms of TLRN, it is straightforward to adapt these networks by backpropagation.

#### 10.4.6. Beyond the Pure Robust Method: Alternatives, Earlier Work, and Research Opportunities

**10.4.6.1. Introduction** The goal of this section was *not* to promote the idea of using the pure robust method in any and all problems of system identification. Up to this point, we have simply tried to prove, with real-world examples, that neural network applications can yield much more useful, practical results if we go beyond the conventional approaches to training or estimation. The pure robust method is a first step in that direction, but there are enormous opportunities available to other researchers who choose to go beyond that first step. It is very sad to see hundreds of papers in the neural network field and the control theory field that contribute relatively little either to improving our arsenal of usable, practical tools, or to deepening our fundamental understanding. At the same time, we see numerous opportunities here—largely neglected—to achieve both things at the same time, simply by paying more attention to certain basic issues.

The main goal of this subsection will be to describe these opportunities in more detail. The key to these opportunities is to *bring together* and *apply* certain ideas that already exist in various strands of control theory, statistics, neural network theory, information theory, epistemology, and elsewhere. A perfect synthesis of these ideas would lead to mathematical algorithms that are impossibly complex to apply in the real world; however, there are plenty of opportunities to proceed in a step-by-step, empirical fashion, applying one idea at a time, improving one's approximations one step at a time, as is common in neural network research and practical engineering. We ourselves do not claim to have extracted the full potential of this approach.

This subsection will proceed as follows. First, for the practical and impatient researcher, we will discuss the compromise method—one step beyond the pure robust method—and the empirical work to date on the compromise and pure robust methods. Second, we will briefly discuss the issue of adaptation in *real time*. Third, we will describe the four successive levels of theoretical analysis that were used to develop the compromise method, and their relations to other issues and methods related to intelligent control:

1. The maximum likelihood approach;
2. The Bayesian (prior probability) approach;
3. The utilitarian (loss function) approach;
4. A modified utilitarian approach.

A deeper, more rigorous reevaluation of this analysis would presumably lead to methods more general and more powerful than the compromise method. Section 10.4.6.4 will describe some preliminary efforts on these lines, and the new approaches that result from them. Likewise, there are a host of similar methods and approaches developed in the field of control theory [31] that could be reevaluated more deeply and extended to the neural network case. Statisticians often disparage the relatively ad hoc, empirically motivated methods one often finds in sources like the *Journal of Forecasting*; however, empirical results of that sort—if analyzed at a deeper theoretical level—could also be very useful in guiding us towards more general methods.

**10.4.6.2. The Pure Robust Method, the Compromise Method, and Earlier Work** The first experiments with the pure robust method were motivated by an attempt to fit a political forecasting model to empirical data back in 1972 [3]. Previous attempts using multiple regression (i.e., conventional maximum likelihood estimation and an ARX model) had led to very poor forecasting results in split-sample multiperiod forecasting. There was reason to believe that the available data on the relevant variables—nationalism and political mobilization data going back up to two centuries—might be subject to some measurement noise. Based on maximum likelihood theory, Werbos fitted a simple vector ARMA (1, 1) model—similar to equations 48 and 49, but with complex nonlinear exogenous variables added as well. Just for the sake of comparison, Werbos also deleted the process noise term ( $e_2$ ) and estimated the “measurement-noise only” model—which is equivalent to the pure robust method. (The vector ARMA model was estimated by backpropagation through time.)

The surprising result was that the pure robust method cut forecast error in half, compared with the ARMA model. From a classical maximum likelihood viewpoint, this shouldn't happen. A more general model—such as the ARMA model—should do at least as well as the more specialized model, at least if the parameters were significant (they were), and if there were some reason to expect the presence of process noise (there was). The ARMA model, by contrast, reduced prediction errors by only about ten percent compared with multiple regression.

To test these results, Professor Mosteller of Harvard recommended a set of simulation studies. Werbos performed multiple simulations of two different dynamic processes, each with six different noise generators attached. The first process was:

$$X(t+1) = (1.03)X(t)(1 + e_1(t)) \quad (59)$$

$$Y(t) = X(t)(1 + e_2(t)) \quad (60)$$

The second dynamic process was similar, except that equation 59 was replaced by:

$$X(t+1) = (.38X(t) + .35X(t-1) + .3X(t-2))(1 + e_1(t)). \quad (61)$$

The six noise processes chosen for  $e_1$  and  $e_2$  involved Gaussian white noise, Gaussian white noise with random outliers in five percent of the cases, and dirty noise with correlations over time. Each combination of noise process and dynamic model (twelve combinations in all) was simulated ten times, yielding simulated time series of two hundred observations each. Broadly speaking, the ARMA model did indeed cut the prediction and estimation errors by a factor of two compared with multiple regression on the simulated data; however, the pure robust method cut the errors in half yet again. The dirtier the system, the bigger the differences; however, even with the simplest systems, the ranking of the methods was still the same [3].

Despite these results, it was obvious, intuitively, that the pure robust method would not be the ideal method to use under *all* circumstances. For example, in equations 59 through 61, there is an underlying exponential trend to the data, which makes it *possible* for a forecasting model to track the overall trajectory of the data over long time intervals. This is not always the case. For example, consider the process:

$$\begin{aligned}X(t+1) &= X(t) + W + e_1(t) \\Y(t) &= (\sin X(t)) + e_2(t).\end{aligned}\quad (62)$$

If the noise,  $e_i(t)$ , is small, it should be possible to track this process by using any model that is capable of outputting a sine wave—but only for a certain amount of time. As the time series grows longer, the random phase shifts will always ensure that the later part of the data will get out of phase with the earlier part, so that comparison between predicted and actual values at large times  $t$  will always show large errors.

In cases like equation 62, we *do not* need to go all the way back to the nonrobust AR (1) or ARMA (1, 1) models. For example, we can try to minimize multiperiod prediction errors, but limit ourselves to a time horizon of  $p$ —a parameter we must pick very carefully. Alternatively—and at lower cost—we can use the compromise method, proposed in [3] as a way to deal with these kinds of situations.

In the compromise method, we minimize a measure of error,  $E_4$ , which is defined by the following system of calculations:

$$\hat{Y}(t) = F(\hat{Y}(t-1), \dots, \hat{Y}(t-p), u, W) \quad (63)$$

$$\hat{Y}_i(t) = (1 - w_i)\hat{Y}(t) + w_i Y_i(t) \quad (64)$$

$$E_4 = \sum_i^T L(\hat{Y}_i(t), Y(t), w), \quad (65)$$

where  $F$  represents the forecasting model or ANN, where  $u$  represents all the various exogenous variables at various times, where  $W$  is the set of model parameters or weights, and where  $w$  (made up of components  $w_i$ ) is a vector of additional parameters or filtering constants. When  $w = 0$ , then the compromise method reduces to the pure robust method. When  $w = 1$ , it reduces to conventional maximum likelihood estimation. (If you feel nervous about your implementation of the compromise method, you can check to make sure it matches the simpler methods in those cases.) A  $w$  of 0.01 would, in effect, yield a delay time of seventy time periods in equation 64, and be very similar to minimizing the 70-period prediction error. If we pick  $w$  so as to minimize the conventional loss function:

$$L_1(\hat{Y}(t), Y(t)) = \sum_i (Y_i(t) - \hat{Y}_i(t))^2, \quad (66)$$

then the compromise method reduces to something like a special case of nonlinear Kalman filtering—which, by its equivalence to ARMA estimation, loses the robustness feature we are looking for. After some contorted analysis, Werbos [37] developed an alternative loss function:

$$L_2(\hat{Y}(t), Y(t), w) = \sum_i \left( \frac{(Y_i(t) - \hat{Y}_i(t))^2}{\sigma_{Y_i}^2 (1 - |w_i|)^2} \right) \quad (67)$$

which led to superior performance on political data that were so noisy that the pure robust method did not work very well. In analyzing these results, Werbos suggested an alternative loss function:

$$L_3 = \sum_i \frac{|Y_i(t) - \hat{Y}_i(t)|}{\sigma_{Y_i}(1 - |w_i|)} \quad (68)$$

In these tests,  $w_i$  was restricted to the interval (0, 1).

The model used in these tests was an upgraded version of a model then used by the Joint Chiefs of Staff for global long-range strategic planning [37]. The compromise method with the alternative loss function led to a reduction of multiperiod prediction error that was only about twenty-five percent for the political conflict variables (such as revolutions in Latin America), which were extremely difficult to predict in any case. It led to a fifty percent reduction in error in predicting GNP, both in the Latin American study and in a follow-up study of U.S. economic data (based on a simple model of the U.S. economy developed by Kuh). Several unpublished student papers at the University of Maryland showed similar reductions of error.

The biggest advantage of the compromise method is its ability to shift smoothly—to a *different* extent for *different* prediction targets—between the extremes of conventional adaptation and pure robust adaptation. In conventional statistics, it is often good enough to give the user two extreme estimation methods and provide guidelines for which method to use when [56]. In intelligent control—where many variables need to be predicted, and the decision needs to be automated and changeable over time—something like the compromise method is essential. *This kind of smooth variation between a classical approach and a worst-case or robust approach can be implemented on many levels in the design of a robust system.*

The loss functions in equations 67 and 68 appear useful, according to the empirical results so far. However, no one has attempted a neural network application as yet. Because of the gross approximations made in deriving these loss functions, there is every reason to believe that you, the reader, could come up with something similar but better—especially if you revisit the theoretical issues to be discussed below. On the whole, equations 67 and 68 appear very effective in nudging the  $w_i$  parameters closer to 1 when they are already close to 1; however, they do not have such a strong effect on values of  $w_i$  which are close to 0. Also, it may be useful to weight the errors for different variables in a different way; for example, for models embedded within an adaptive critic control system, one might weight the error in predicting  $Y_i$  according to the RMS average of the derivative of  $J$  with respect to  $Y_i$ .

**10.4.6.3. Real-time Learning** Section 10.8 will describe how to implement the compromise method in an off-line mode, using backpropagation through time. *All* of the models and identification methods in this chapter boil down to minimizing an error function over time, in the presence of time lags. To implement these methods, one basically needs to calculate the derivatives of error with respect to the weights or parameters. Therefore, for real-time adaptation, one basically needs a method that moves *forwards* in time, to calculate the same derivatives that are normally calculated by backpropagation. (In some real-time applications, it may be useful to develop an off-line model first, in order to develop useful hidden variables; these variables, in turn, can be input to a more static real-time network.)

One can calculate the derivatives exactly in forwards time by using the forwards perturbation technique [20], which was later called the “Williams-Zipser” method and “dynamic backpropagation”; however, the resulting calculations are  $N$  times as expensive as backpropagation through time. If  $N$  is small, this may be useful. If it is large—as in biological systems—the method becomes hopelessly unrealistic. See Chapter 5 for more details. Section 10.6 may help in using [20].

Alternatively, one can simply truncate the derivatives, as suggested by equation 25; however, this dramatically reduces robustness over time because it totally ignores the connections over multiple time intervals.

Finally, one can approximate the derivatives in a consistent way by using the Error Critic design, presented in Chapter 3. The present chapter gives a variety of error functions different from the example in that chapter, but the extension to alternative error functions and functional forms is straightforward. We would speculate that the human brain itself is based on such an extension.

#### 10.4.6.4. Basic Principles and Derivation of the Loss Function for the Compromise Method

Once again, the compromise method evolved in response to empirical results. The approximations used to derive it were very drastic and can be improved upon (or made more adaptive). Before understanding the robust approach, theoretically, it is necessary to review some earlier approaches, which will still be important to future research.

The early work by Werbos [3] was initially based on the classical Bayesian approach to parameter estimation. In that approach, one begins by formulating a *stochastic* model, such as equations 48 and 49 or equations 50 and 51. (Nearest-neighbor systems or spline models can also be formulated as stochastic models, but it takes some care to avoid logical inconsistencies.) One tries to find that set of parameters,  $W$ , that have the highest probability of being the true set of parameters, based on what we know *after* accounting for the available data; in other words, we try to maximize the conditional probability  $Pr(W|data)$ .

The first deep problem for theorists in parameter estimation is the fact that  $Pr(W|data)$  is not really well defined in an objective manner. From Bayes' law, it is well known that:

$$Pr(W|data) = \frac{Pr(data|W)Pr(W)}{Pr(data)} \quad (69)$$

On the right-hand side, only one term, the “likelihood term”— $Pr(data|W)$ —is well specified, for any well-specified stochastic model. We can ignore the term  $Pr(data)$  because it does not affect our maximization with respect to  $W$ . The third term— $Pr(W)$ —represents our *prior* assessment of the probability of different parameters, our assessment *before* any data become available. No matter how large our database, this term is still present, and it has been a source of deep and persistent confusion even in practical applications.

In the maximum likelihood approach—as practiced by most engineers—one tries to be “scientific” by assuming that all sets of parameters are equally likely *a priori*. This simply wipes out the  $Pr(W)$  term. By maximizing the likelihood term itself, one has a direct recipe for calculating what appears to be an optimal estimate of all parameters. Given a perfect model and an infinite database (with enough variety of the input variables), this approach will eventually converge on the optimal estimates.

Practical users of statistical methods, beset by limited data, usually recognize the need to account for specific, prior information they have about the variables they are trying to predict. For example, econometricians have long grappled with the problem of how to reconcile empirical, statistically based models with first-principle, engineering-based models [56]. Typically, econometricians do *not* try to quantify their prior probabilities,  $Pr(W)$ , and use a Bayesian regression routine. They use their prior information to guide them in setting up alternative stochastic models; they use it again, *after* performing maximum likelihood estimation, to select the final model and to “tweak” its parameters [56]. Methods like backpropagation could be used as part of this process to adapt or calibrate parameters in complex first-principle models in a more automated way.

There is *another* kind of prior information, above and beyond our knowledge of specific variables, that is extremely important to fields like neural networks and intelligent control. To explain this crucial point, we will begin with an example.

In the neural network field, we often try to “learn” the mapping from a vector  $X$  to a scalar  $Y$ . Suppose, for example, that the vector  $X$  had twenty different components, and that we can distinguish ten possible values for each of these twenty variables. Thus, to specify  $Y$  as a function of  $X$ , we would need to know the values of  $Y$  for each of  $10^{20}$  different points. Even if we do *not* assume the possibility of noise, we would need a huge amount of data even to begin learning the mapping. It is this kind of impossible-looking problem that makes the purist variety of theoretician give up and hide, thereby leaving the deep and important theoretical questions to people like engineers.

To cope with these kinds of practical problems, we need to exploit the fact that *not all functions are equally likely a priori*. The real world tends to have certain properties of smoothness, of similar values in nearby points, of sparse connectivity, of symmetry, of global order, and so on. In the real world, simpler models (or, equivalently, models with a lot of the parameters zeroed out) are more likely to be true or robust than are complex models. Deep philosophers have recognized for centuries that this principle—Occam’s Razor—is essential to human learning and induction. (Indeed, one can argue that Occam’s work had a very substantial effect on subsequent human history, particularly on science and religion.)

Our first deep challenge, then, is to formulate Occam’s Razor in a more operational way, which is truly useful in adapting ANNs (or other models) in the real world. The Razor must be sharp enough to prevent floundering with overly complex models, but dull enough to allow our system to be truly “open-minded”—able to adapt to a wide variety of possible environments. Good ANN researchers already know that simpler nets lead to better generalization, but we need to make this knowledge firmer and more operational.

This challenge was met successfully in the field of classical AI by the work of Solomonoff in the early 1960s [57]. Solomonoff proposed that the *a priori* probability for any model should be  $e^{-kS}$ , where  $k$  is some constant and  $S$  is the number of symbols required to describe the model *as a program in some kind of basic Turing machine*. Because any Turing machine can be programmed to mimic any other kind of Turing machine, with a *finite* program (using, say,  $S_0$  symbols), we are *guaranteed* that such a system could “learn” to change its language after acquiring only  $S_0$  pieces of information. In effect, Solomonoff proved that such a system would be invariant to the initial choice of modeling language; thus, it would be truly open-minded in an important way. Solomonoff’s work is of enduring importance, even to ANN research, but—like much of classical AI—it mainly focuses on *Boolean variables*. For our purposes, we need to extend the concept further to deal with models of continuous

variables. We need to account more concretely for phenomena like local smoothness, forecasting by analogy, and so on.

In neural network theory, it is important that the multilayer perceptron (MLP) does *not* live up to Solomonoff's notion of a Turing machine or Turing language. A Turing machine has the ability to write out to a kind of "blackboard" or "tape," which it can later read back and reanalyze, all as part of the calculations it makes before outputting a single forecast. The simultaneous-recurrent network does have the ability to perform multiple calculations, inputting its own outputs, as part of a single computation cycle (see Chapter 13); therefore, it is very close to fulfilling Solomonoff's requirement. It is not quite perfect, because the number of cells in the network is finite, whereas a true Turing machine is allowed an infinite blackboard. To *completely* live up to Solomonoff's results, a system must be able to engage in overt symbolic reasoning with the aid of unlimited paper and computer support; this kind of symbolic reasoning can be *learned* by neural networks (e.g., the networks in the brain), but it may well be impossible to implement it on a lower level as part of the neuronal hardware [15].

When we insert the Solomonoff priors back into equation 69, and take logarithms on both sides, we end up with an interesting recipe for parameter estimation. We are asked to minimize the sum of two terms: (1) the log likelihood, which can be described as the residual entropy of the database after the model is available; and (2) the  $kS$  term, the entropy or information content of the model. This provides a rational basis for the notion of minimizing "total entropy" or "total complexity," a well-known approach in information theory. Theorists like Solla [58] and Kazakos [59] have tried to apply this approach in the neural network field; however, Kazakos has stated that there are many insights and ideas about complexity measures and robustness in journals like the *IEEE Transactions on Information Theory* that have yet to be fully used or appreciated in system identification (e.g., see [58]).

Economists and statisticians have also developed formulations of Occam's Razor for use in the linear case. Economists often find themselves with data on a very limited number of time points, for a very large number of variables. They have been forced to develop ways of coping with this problem. There are two major techniques in common use:

1. Ridge regression, which effectively penalizes large estimates of parameters
2. Stepwise regression (by hand or automated), in which parameters are deleted (zeroed out) when they fail a basic test of "statistical significance"

The well-known statistician Dempster [60] has given a thorough review of these techniques and their many (less effective) competitors. Most of them have analogues in the neural network literature.

In ridge regression, one assumes that the *parameters themselves*— $W$ —come from some sort of Gaussian distribution. Instead of assuming *a priori* that the variance of that distribution is *infinite* (as in maximum likelihood theory), one tries to *estimate* the variance of this distribution. This is called the "empirical Bayes" approach. Clearly, this general flavor of approach could be used more generally to help us generate prior probabilities of ANN adaptation. In practice, ridge regression asks us to minimize an error function:

$$\sum_i (e(i))^2 + k \sum_{ij} (W_{ij})^2, \quad (70)$$



where  $k$  is another parameter to be estimated in another way [60]. It is straightforward to apply this method directly in ANN research (as first suggested in [34]), but it may be possible to do better yet. One can argue that equation 70 is too "closed-minded," in a sense, about large parameters; that it does not adequately value the zeroing out of parameters (which is also important [60]); and that it does not reflect the tendency for the weights affecting a given target to vary considerably in size. To account for such concerns, one might, instead, assume a *lognormal* probability distribution, which leads to an error function:

$$\sum_i (e(t))^2 + k \sum_{ij} \log \frac{1}{|W_{ij}|} . \quad (71)$$

This kind of error function reflects the usual practice in econometrics of zeroing out parameters that are only one standard deviation or so away from zero. The constant  $k$  can be adapted by techniques similar to those of Su, described earlier in this section. Because the term on the right has a powerful effect in zeroing out some weights, one must take care in developing initial weights, as with the other methods in this chapter. In the limit [34], one might give greater credence (a smaller  $k$ ) to connections between neurons in similar "clusters," etc. In the ANN field, there is recent work on input compression, by Cooper, and on B-spline methods, which may be significant and novel here.

In any event, the statistical profession has firmly established the validity of minimizing modified error functions that include a direct reference to the parameters or weights of a model.

Unfortunately, the sophisticated Bayesian or Occamite approach above was still not enough to explain the empirical results discussed in section 10.4.6.2. ARMA models were estimated with significant  $t$ -ratios and  $F$ -ratios, which suggested that the full model should be used in forecasting; however, the pure robust method still did better, both in multiperiod forecasting *and* in parameter estimation, with empirical *and* simulated data.

To explain or exploit these empirical results, one must move up to a more sophisticated theoretical approach—the "robust estimation" approach [30].

In robust estimation, we give up the assumption that our original stochastic model will in fact be "true," in some objective sense, for some set of values for the parameters  $W$ . We assume that our stochastic model will typically (though not always) be a highly simplified description of a more complex reality, regardless of whether that model is an ANN or a first-principle model. We try to adapt or estimate that model so that it will do a good job of forecasting (or control) *despite* that limitation.

Some theoreticians have argued that robust estimation should never be necessary. If our stochastic model is not good enough—as determined by *any* objective measure of how the model fits the data—we should simply change it until it *is* good enough. In actuality, the real process we are trying to identify may simply be too complex to make this a workable strategy. If there are *many, many* objective measures available, the complexity required to account for all of them may make it difficult to avoid large estimation errors. Still, this approach should not be discarded altogether. In fact, the pure robust method could be viewed as one way of *implementing* that approach. The pure robust method may be seen as a kind of complex objective measure or diagnostic used to evaluate a stochastic model; if we find that it is *possible* to do significantly better in multiperiod forecasting than the original model suggested, we may then choose to stay with the pure robust (or compromise) method. This kind of reasoning may well be useful as a guide to tuning the filtering parameters,  $w_i$ , in the

compromise method or in various extensions of the compromise method. Still, we have to ask *what to do* when our diagnostic does tell us that better long-term forecasting is possible. Direct use of the robust methods is the obvious choice. We wouldn't want to add a lot of terms to the old stochastic models to make them fit better, simply because those models did, in fact, forecast poorly over long time intervals. There may be other possible responses, however, that may merit further research.

In robust estimation, our biggest challenge is to find workable ways of estimating a model that do not assume that its stochastic version is true. There is a huge literature on how to do this when a stochastic model is *almost* true, when we assume that the equations of the model are true *but that* the noise may not come from a Gaussian distribution. Those methods still assume that the noise is "clean" (i.e., uncorrelated). For the more general case, we are aware of only two workable approaches to robust estimation: (1) a utilitarian-based approach; (2) an approach based on minimizing parameter errors. From an Occamite point of view, the best procedure would be to always consider the set of *all possible* models, instead of just one model; however, even if we do maintain a handful of alternative models, we still need to use robust estimation techniques for each of that handful. (In effect, each explicit model must be used to integrate or represent a large set of models we cannot afford to be explicit about.)

The utilitarian approach has been around for a very long time. As an example of this approach, suppose that we are trying to forecast a single variable. Suppose that we are trying to predict the savings rate,  $s$ , of different households, as a function of income and other variables. The savings rate is defined as total savings ( $S$ ) divided by disposable income. We expect that the noise in this process will not be a function of income or other variables; thus, for people earning \$1,000,000 per year, our forecast may be off by ten percent or so—\$100,000—but for people earning \$10,000 or so we expect errors on the order of \$1,000. A maximum likelihood statistician would tell us to minimize the error function:

$$E = \sum_i (s(t) - \hat{s}(t))^2. \quad (72)$$

A utilitarian economist might instead try to minimize the loss function:

$$L = \sum_i (S(t) - \hat{S}(t))^2. \quad (73)$$

The utilitarian would argue that the *cost* of an error (in many applications of this model) would depend on the *size* of the error in dollars; this justifies giving greater weight to observation where more dollars are involved. He would argue that there are two sets of parameters out there in the real world—the parameters that minimize  $E$  and the parameters that minimize  $L$ ; for his application, it is  $L$  that matters, so he prefers to use a procedure that is *consistent* with his application. A maximum likelihood purist would reply that there is only *one* true set of parameters; minimizing equation 73 is simply a less *efficient* way of estimating the same set of parameters, which can be estimated more *efficiently* by minimizing equation 72. In an ideal world, one might try it both ways, and then try to see if there really were a measurable difference between the two sets of parameters; however, this judgment would be affected somewhat by one's *prior* expectations about the possibility of such a difference.

There is a simpler example of this same principle, which has long been familiar to statisticians. Suppose that we are trying to estimate the median of a random distribution. To be perfectly *consistent*, we can simply use the median of our sample data. However, if we have good reason to believe that our data comes from a Gaussian distribution, we can use the sample *mean*; this is a more efficient estimator, and we know that the median is the same as the mean anyway, *if* we are sure of the distribution. Using the sample median is essentially the robust or utilitarian way to go. There are other possible estimators, of course.

In our application, the utilitarian approach seems very straightforward, at first. In Model-Predictive Control, our payoffs depend on the multiperiod forecasting ability of our model or ANN. The pure robust method minimizes those errors directly. Therefore, it is consistent with what we need, and we should use *it* rather than the maximum likelihood method.

In actuality, there are two important complications here. First of all, our errors in control do not depend in such a simple way on the sum of prediction errors over all possible prediction intervals; the literature on control theory [31–33] suggests a more complicated relation, related to our earlier suggestions for a moving-window approach and for the compromise method. Secondly, our earlier example of a phase-shift process shows that the pure robust method may in fact be *extremely inefficient* in estimating parameters in some situations. Loss functions were developed for the compromise method in 1977 [36] that failed very badly in empirical tests, derived from a pure utilitarian approach.

The alternative loss functions shown in section 10.4.6.2 were derived from a modified version of the utilitarian approach [61]. The idea was to minimize the long-term prediction errors, *as predicted* by the model itself, but *assuming the worst* about the accumulation (i.e., autocorrelation) of error. There were gross approximations involved, but the method has proven useful.

To begin with, Werbos [61] considered the case of a simple univariate forecasting model with the compromise method, deriving:

$$Y(t) = \hat{Y}(t) + e(t) \quad (74)$$

$$\bar{Y}(t) = (1 - w)\hat{Y}(t) + wY(t) = \hat{Y}(t) + we(t) \quad (75)$$

$$\hat{Y}(t+1) = \theta\bar{Y}(t) = \theta\hat{Y}(t) + \theta we(t) \quad (76)$$

$$Y(t+1) = \theta\hat{Y}(t) + \theta we(t) + e(t+1). \quad (77)$$

And likewise:

$$Y(t+n) = \theta^n \hat{Y}(t) + e(t+n) + \theta we(t+n-1) + \theta^2 w^2 e(t+n-2) + \dots \quad (78)$$

The error terms on the right-hand side of equation 78 represent the errors in multiperiod prediction. Assuming the worst about the correlation between errors over time, but assuming that  $\theta w$  is less than one (as it would be in any stationary process, when we enforce the rule that  $w$  can be no larger than one), the expected value of the square of the sum of these error terms will be:

$$\frac{\sigma_e^2}{(1 - |\theta w|)^2} \quad (79)$$

where  $\sigma_e^2$  is the variance of the error term,  $e(t)$ . For a simple univariate model, we would then pick  $w$  so as to minimize this measure of error. For a multivariate model or an ANN, where there is no scalar  $\theta$  available, we assumed that errors also accumulate across variables in the worst possible way, and assumed a  $\theta$  of 1; this led to the loss functions of section 10.4.6.2. Again, one could easily do better.

Starting in 1981, we have explored yet another fundamental approach, mentioned briefly in [35]. This approach may be called the  $\sigma_w$  approach. In the  $\sigma_w$  approach, we go back to the maximum likelihood idea that there is only *one* set of parameters that is best, both for short-term forecasting and for long-term forecasting. Clearly, this will need to be reconciled with the utilitarian approach, eventually, but so far it seems to follow the empirical evidence a bit better; in predicting conflict in Latin America [37], Werbos found that the compromise method did better in *all* prediction intervals in split-sample tests. In a sense, this is like the usual robust approach, where we accept the model at face value, and assume that the problem lies in the dirty noise; however, in this case, we allow for correlated and truly dirty noise, not just non-Gaussian distributions.

The idea is to pick those values of the filtering parameters,  $w$ , which lead to the smallest possible estimation errors in those parameters— $W$ —which will actually be used in the control application. Instead of using the maximum likelihood estimates of the estimation errors, we would like to use more robust estimates, such as the bootstrap estimates of Friedman or the empirical estimates of Hjalmarrsson and Ljung [62]; however, we need to use simple approximations, in the end, to fit with the neural network rule of limiting ourselves to inexpensive calculations.

Once again, we have begun our analysis in the univariate linear case. In that case, the usual measure for standard error may be written as:

$$\sigma_0^2 = \sigma_e^2 / \left( \frac{\partial^2}{\partial \theta^2} \sum_i (e(t))^2 \right) \quad (80)$$

This is similar to equation 79, except that the denominator is somewhat different; thus, our prior method was somewhat similar to picking  $w$  to minimize  $\sigma_0^2$ . To evaluate the denominator, note that:

$$\frac{\partial^2}{\partial \theta^2} \sum_i (e(t))^2 = \sum_i e(t) \frac{\partial^2}{\partial \theta^2} e(t) + \sum_i \left( \frac{\partial}{\partial \theta} e(t) \right)^2 \quad (81)$$

Note that the leftmost term on the right-hand side represents the correlation between error and another quantity; in general, there is little reason to expect a strong correlation, and—in a forecasting situation—we always try to estimate parameters so that previous variables correlate very little with error at time  $t$ . Thus, for a crude, preliminary analysis, we will approximate that term as zero; we will use the rightmost term of equation 81 as our denominator.

Simple substitutions yield:

$$e(t) = Y(t) - \hat{Y}(t) \quad (82)$$

$$\hat{Y}(t) = \theta Y(t-1) = \theta(1-w)\hat{Y}(t-1) + \theta w Y(t-1). \quad (83)$$

Mathematical induction on equation 83 yields:

$$\hat{Y}(t) = \theta w(Y(t-1) + \theta(1-w)Y(t-2) + \theta^2(1-w)^2Y(t-3) + \dots). \quad (84)$$

If we assume the worst about correlations between  $Y$  values over time, we arrive at the following estimate of the RMS average size of equation 84:

$$\sigma_{\hat{Y}} \approx \frac{\theta w}{1 - \theta(1-w)} \sigma_Y. \quad (85)$$

Going back to equations 81 and 82, we may approximate our denominator as:

$$\sum_i \left( \frac{\partial}{\partial \theta} \hat{Y}(t) \right)^2 \approx \sum_i \left( \frac{\partial}{\partial \theta} \left( \frac{\theta w}{1 - \theta(1-w)} \sigma_Y \right) \right)^2, \quad (86)$$

which reduces to:

$$\left( \frac{w \sigma_Y^2}{(1 - \theta(1-w))^2} \right)^2. \quad (87)$$

Substituting this into equation 80, and ignoring the  $\sigma_Y^2$  term (which has no effect on our choices of  $w$  and  $\theta$ ), we arrive at a different kind of loss function to minimize:

$$L_5 = \left( \sigma_e \frac{(1 - \theta(1-w))^2}{w} \right)^2. \quad (88)$$

In cases where  $\theta$  is near zero, and larger values of  $w$  lead to smaller  $\sigma_e$ , this tells us to pick  $w$  as large as possible; since  $w = 1$  is the largest allowed value (representing the maximum likelihood case), we pick that value. In fact, this still applies for *any* value of  $\theta$  less than one-half. However, when  $\theta$  is near 1, this tells us to minimize  $\sigma_e w$ , which tends to push very strongly for the pure robust method.

This analysis gives us considerable insight into the proper use of the compromise method. However, it is not clear what to do about this insight in neural network applications. As a first crude approach, we might "approximate"  $\theta_i$  as the square root of  $1 - (\sigma_e^2 / \sigma_Y^2)$ , as suggested in [61]; however, this is so crude that its value is questionable at present. A very crude, multivariate linear version of the reasoning above leads to a denominator more like:

$$TR((I - \theta(I - W))^T W^T Q W (I - \theta(I - W))^{-2} Cov(Y)) \quad (89)$$

where  $\theta$  is now a matrix, where  $W$  is a filtering matrix, where  $Q$  is a positive metric representing the cost of different kinds of error, and  $\text{Cov}(Y)$  is the variance-covariance matrix of  $Y$ , assuming all variables have been reduced to a mean of zero. If  $\theta$  is close to being *any* kind of unitary matrix, then the optimal value of  $I - W$  is not  $I$  (as in the pure robust method), but  $\theta^{-1}$ . This suggests two approaches to getting maximum robustness from a complex model (such as an ANN model): (1) actually use a complete filtering *matrix* or network; (2) try to transform the representation, so that individual variables tend to represent eigenvectors of the dynamic matrix  $\theta$  or other such stable variables.

Both of these approaches have some hope for them, but are completely new and unexplored in this context. The network-filtering approach leads to the obvious problem of how to minimize a rather complex error function. With simultaneous-recurrent circuits looping around a network, it is feasible to calculate inverses (or, more precisely, to multiply the inverse of a network by a vector), but there is some major complexity here and issues about numerical stability and consistency. The eigenvector approach would be very similar in spirit to the encoder/decoder schemes that are widely used in neural network research; for example, see the Stochastic Encoder/Decoder/Predictor design in Chapter 13. Likewise, variables that have big  $J$  derivatives (in an adaptive critic design) also tend to be relatively stable variables. In a network with sticky weights (as discussed in section 10.4.6.2), the sticky weights themselves may be used as estimates of  $\theta$ . Clearly, it will take some creativity to get all these pieces together in a logical manner, but the pieces do exist.

Taking this all a step further, the discussion of "chunking" in Chapter 13 does include a discussion of further issues of relevance to robustness. Likewise, it would seem logical to modify our estimation of some components of  $W$ —such as sticky weights—in order to improve *their* influence on estimation errors of *other* components of  $W$ .

There is a rich field here for future research—both empirical and theoretical—of enormous importance to practical success in intelligent control.

## 10.5. A METHOD FOR PRUNING ANNS

Stringent pruning is often vital to ANN performance, for reasons discussed at length in this chapter. At least two other pruning or sizing methods were discussed in section 10.4. The method described here was developed in 1990 by Bhat and McAvoy [2]. This method has been applied successfully to real-world problems involving felt design; however, [2] mainly discusses applications to simulated systems and to the pH/CSTR example of section 10.3.

In reviewing the background, Bhat and McAvoy stressed the need for obtaining a good training set as well, using classical notions of experimental design [63] to ensure variance of the inputs. They reviewed earlier methods, based on deleting *entire neurons*, which have a number of limitations and, in any case, do not provide all the power we need by themselves.

The Bhat/McAvoy algorithm uses *two* evaluation functions for the ANN, one of them ( $L_w$ ) used to adapt the weights, and the other ( $L_{NN}$ ) used to evaluate the ANN after training. They define:

$$L_w = \left( \sum_i^n (\hat{Y}_i - Y_i)^2 \right) + \lambda * \text{COMPLEXITY} \quad (90)$$

where:

$$COMPLEXITY = \sum_k N_k \sum_j^{N_k} W_{jk}^2 \quad (91)$$

and  $N_k$  is the number of active, nonzero weights arriving at each node; and (for the case  $n = 1$ ):

$$L_{NN} = \sigma_e^2 + \sigma_Y^2 \frac{k}{T} \quad (92)$$

where  $T$  is the number of observations in the training set,  $k$  is the number of weights in the network, and  $\sigma_e^2$  and  $\sigma_Y^2$  are the sample variances of the error and target variables, respectively. (The sample variance of  $Y$  is defined as the average observed value of  $(Y - \mu_Y)^2$ , where  $\mu_Y$  is the mean of  $Y$ .)

The algorithm itself is similar to an earlier one proposed by Niida et al. [64]:

1. Start with a complex MLP full of weights and adapt the weights so as to minimize  $L_W$  with  $\lambda = 0$ .
2. Increment  $\lambda$  by a small amount, and adapt the weights again to minimize  $L_W$ .
3. Delete (strip) all the weights in the dead zone around zero.
4. After removing the stripped weights (i.e., holding them to zero), adapt the network again with  $\lambda = 0$ .
5. Evaluate  $L_{NN}$  for the resulting network (and store  $L_{NN}$ ).
6. If the current  $L_{NN}$  is smaller than that obtained before, with a smaller value of  $\lambda$ , go back to step two and try again.
7. If the current  $L_{NN}$  is larger, then go back to the previous version of the ANN (for a smaller  $\lambda$ ), and use that as the final version. Quit.

Niida et al. had proposed a different measure of complexity:

$$COMPLEXITY = \sum_k |\prod_j W_{jk}|,$$

but this failed in all of our tests because of high sensitivity to changes in  $\lambda$ , difficulties in handling weights that vary by orders of magnitude, and erratic behavior when small weights are deleted. The measure of  $L_{NN}$  used here was based on work by Barron for models that are *linear* in their coefficients [65]; therefore, there may be some room for improvement in the procedure.

## 10.6. HOW TO USE THE CHAIN RULE FOR ORDERED DERIVATIVES

The chain rule for ordered derivatives ("backpropagation") is a general-purpose tool for calculating derivatives efficiently through *any* nonlinear, sparse system. As a rule of thumb, if it takes  $N$  calculations to run a model, it should take only  $kN$  calculations to extract the derivatives of any target variables (such as error or utility) with respect to *all* of the inputs and parameters of the model, in a *single* sweep backwards through the model. As computers grow more powerful and the problem addressed grow correspondingly larger, the efficiency of derivative calculations will become even *more* important, because the ratio between  $N$  calculations and  $N^2$  calculations will grow larger.

This section will *only* discuss the problem of calculating derivatives through an *ordered* system. Roughly speaking, an ordered system is one where we have a definite, fixed procedure for calculating the outputs as a function of the inputs; it differs from a "relaxation system," which requires the solution of a set of nonlinear simultaneous equations in order to calculate the outputs. Chapter 13 describes how to calculate derivatives effectively through a relaxation system (a "simultaneous-recurrent network"), *once* you know how to calculate through an ordered system.

The section will proceed as follows. First, it will describe what an ordered derivative is and give the chain rule. Second, it will give an example of how to apply it. (The procedure here could easily be automated.) Finally, it will describe the notion of a "dual subroutine," which is crucial to the use of this chain rule within larger, more sophisticated control applications. The material here is mainly taken from [4] and [22].

### 10.6.1. Ordered Systems and Ordered Derivatives

An ordered system is a system made up of variables or quantities that we can calculate in a definite order. The two most common examples are static systems and dynamic systems. An example of a static system might be:

$$z_2 = 4 * z_1 \quad (93)$$

$$z_3 = 3 * z_1 + 5 * z_2. \quad (94)$$

More generally, a static system might be:

$$z_{m+1} = f_{m+1}(z_1, \dots, z_m) \quad (95)$$

$$z_{m+2} = f_{m+2}(z_1, \dots, z_{m+1})$$

$$\vdots$$

$$TARGET = z_N = f_N(z_1, \dots, z_{N-1}),$$

where  $z_1$  through  $z_m$  are the parameters and inputs, and where  $f_{m+1}$  through  $f_N$  are differentiable functions. When we are seeking the derivatives of a particular quantity,  $z_N$ , we usually describe the



system so that  $z_N$  is the *last* quantity to be calculated; it is convenient to give that quantity a special name ("TARGET").

In some ordered systems, we can still calculate the desired output ( $z_N$ ) after changing the order of calculation; however, this merely gives us a choice of alternative ordering schemes, each of which is a valid starting point. (For those who are interested in parallel computing, however, it may be interesting to note that our derivative calculations will preserve the lattice structure—the parallelization—of the original model.) The *key requirement* is that we can calculate any quantity,  $z_i$ , when we know the values of preceding quantities.

An example of an ordered dynamical system would be:

$$Y_2(t) = W_1 * Y_1(t) + W_2 * Y_1(t-2) \quad (96)$$

$$\star Y_3(t) = 3 * Y_1(t-1) + 4 * Y_2(t) * Y_2(t-1) \quad (97)$$

$$E(t) = E(t-1) + (Y_3(t) - Y_3^*(t))^2, \quad (98)$$

where  $E(t)$  is a kind of error measure. In actuality, this dynamic system may be treated as a *special case* of equation 95, where we define  $z_1$  as  $W_1$ ,  $z_2$  as  $W_2$ ,  $z_3$  as  $Y_1(1)$ , ...,  $z_{T+2}$  as  $Y_1(T)$ ,  $z_{T+3}$  as  $Y_2(1)$ ,  $z_{T+4}$  as  $Y_3(1)$ , and so on. Using this kind of approach, Werbos has applied backpropagation even to models with *two* time dimensions, as required by some economic models of the natural gas industry [4]; however, this section will not give explicit attention to that kind of model.

The chain rule for ordered derivatives may be written (for the system in 95):

$$\frac{\partial^+ \text{TARGET}}{\partial z_i} = \frac{\partial \text{TARGET}}{\partial z_i} + \sum_{j=i+1}^{N-1} \frac{\partial^+ \text{TARGET}}{\partial z_j} * \frac{\partial z_j}{\partial z_i}, \quad (99)$$

where the derivatives with the + superscript represent *ordered* derivatives, and the derivatives without superscripts represent simple partial derivatives. To use this chain rule, you need to know only two facts:

1. That the ordered derivatives of *TARGET* with respect to an input or parameter do, in fact, represent the effect of changing that input on changing *TARGET*; these are the appropriate derivatives to use in error minimization, utility maximization, and so on.
2. That the "simple partial derivatives" in equation 99 are worked out simply by differentiating *that equation in the model* that defines  $z_j$  (or *TARGET*) as a function of earlier variables.

The full definition and proof of equation 99 was given in [3] and reprinted in [4]. To use equation 99 in practice, you must start with the case  $i = N - 1$  and work backwards to  $i = 1$ .

It is usually best to *test* these derivatives by perturbing inputs and making sure that the calculated derivatives are right. When they aren't, one can narrow down the problem by testing the validity of *intermediate* ordered derivatives. One can perturb an intermediate quantity (or equation)  $z_j$ , and *then* run the rest of the model (from  $j + 1$  to  $N$ ) in normal fashion; this should change  $z_N$  in proportion to

the ordered derivative of  $z_N$  with respect to  $z_j$ . If it does not, you then know that there is a downstream problem, in calculating the ordered derivatives of  $z_N$  with respect to  $z_j, z_{j+1} \dots z_{N-1}$ .

In the example of equations 93 and 94, the simple partial derivative of  $z_3$  with respect to  $z_1$  (the *direct* effect of changing  $z_1$ ) is 3. The ordered derivative (the *total* effect) is 23, because of the *indirect* effect by way of  $z_2$ .

### 10.6.2. Implementation Procedure

Section 10.6.1 includes all the information you need, in principle, to apply the chain rule for ordered derivatives. However, there is a straightforward way of applying that chain rule, given at length in section IV of [4], which could be used by people who find it difficult to understand section 10.6.1. This straightforward procedure is useful even for the expert in disentangling complex systems.

In this procedure, we begin by defining an abbreviation:

$$F_{-z_i} \triangleq \frac{\partial^* TARGET}{\partial z_i}. \quad (100)$$

In most applications, this abbreviation gives us all the information we need, because there is only one *TARGET* variable (like error). Also, the abbreviation can be used directly to naming variables in computer code. In more complex applications, this kind of abbreviation is even more essential. Intuitively,  $F_{-z_i}$  may be thought of as "the feedback to variables  $z_i$ ."

In this notation, equation 99 may be written:

$$F_{-z_i} = \frac{\partial^* TARGET}{\partial z_i} + \sum_{j=i+1}^{N-1} F_{-z_j} * \frac{\partial z_j}{\partial z_i}. \quad (101)$$

The summation on the right-hand side appears complex at first; however, since the equation determining  $z_j$  will usually refer to only a few other variables  $z_i$ , the simple partial derivative of  $z_j$  with respect to  $z_i$  is zero in most cases. In most cases, the summation boils down to a sum of only one or two nonzero terms.

Step one in our procedure is to *spell out, in proper order, all of the calculations* used to calculate *TARGET*. Equations 96 through 98 are an example, if we add just one more equation:

$$TARGET = E(T). \quad (102)$$

Because dynamic systems are harder to work with than static systems, we will focus on that example of an ordered system in this section.

After spelling out the forwards system (equations 96, 97, 98, and 102), our job is simply to work out the backwards equations that will be used to calculate the required derivatives. In working out these equations, we first write down the equation for  $F_{-}(Z)$ , where  $Z$  is the *last* variable calculated in the forwards system; then we work out the equation for the variable before that, and so on. By writing down the equations in this order, we can be sure that we can actually *use* them in this order in our computer program.

In the forwards system, in our example, the *last* variable to be calculated was  $E(t)$ , the variable on the left-hand side of the last equation before the definition of the *TARGET*. We want to work out the equation to calculate  $F\_E(t)$ , for all  $t$ . Here we find that we face two different conditions, depending on whether  $t = T$  or  $t < T$ . For  $t = T$ ,  $E(t)$  affects *TARGET* directly but does not affect any other variables; thus, we have:

$$F\_E(T) = \frac{\partial \text{TARGET}}{\partial E(T)} = 1, \quad (103)$$

which follows from differentiating equation 102.

For  $t < T$ , we look for occurrences of  $E(t)$ —lagged or unlagged—across the right-hand sides of *all* the equations in the forwards system. (In other words, we are looking for quantities  $z_j$  such that  $\partial z_j / \partial E(t)$  is nonzero.) There is only one such occurrence here, located in equation 98; therefore, the sum on the right-hand side of 101 has one nonzero entry in this case. Substituting into equation 101, we get:

$$\begin{aligned} F\_E(t) &= \frac{\partial \text{TARGET}}{\partial E(t)} + F\_E(t+1) * \frac{\partial E(t+1)}{\partial E(t)} \\ &= 0 + F\_E(t+1) * 1 = F\_E(t+1). \end{aligned} \quad (104)$$

To simplify our computer program, we can combine 103 and 104 to get:

$$F\_E(t) = 1. \quad (105)$$

Now that we have calculated the feedback to  $E(t)$ , which was calculated in equation 98, we go back to the previous equation, 97. We try to work out the equation for  $F\_Y_3(t)$ . Looking across the entire system again, we see the  $Y_3$  appears on the right in only one equation, equation 98. Thus, we deduce:

$$\begin{aligned} F\_Y_3(t) &= \frac{\partial \text{TARGET}}{\partial Y_3(t)} + F\_E(t) * \frac{\partial E(t)}{\partial Y_3(t)} \\ &= 0 + 1 * (2(Y_3(t) - Y_3^*(t))) = 2(Y_3(t) - Y_3^*(t)). \end{aligned} \quad (106)$$

Then we go back to calculating the feedback to  $Y_2(t)$ , which appears *twice* in equation 97. For the case  $t < T$ , we derive:

$$\begin{aligned} F\_Y_2(t) &= \frac{\partial \text{TARGET}}{\partial Y_2(t)} + F\_Y_3(t+1) * \frac{\partial Y_3(t+1)}{\partial Y_2(t)} + F\_Y_3(t) * \frac{\partial Y_3(t)}{\partial Y_2(t)} \\ &= 0 + F\_Y_3(t+1) * (4Y_2(t+1)) + F\_Y_3(t) * (4Y_2(t-1)). \\ &= 4F\_Y_3(t+1) * Y_2(t+1) + 4F\_Y_3(t) * Y_2(t-1). \end{aligned} \quad (107)$$

Note that  $F_{Y_3}(t)$  was complicated enough that there is no value in substituting in from equation 106; it is more efficient for the computer to calculate equation 106 as such, and then move on to 107. Also, equation 15 can be used for the case  $t = T$  as well, if we adopt the convention that  $F_{Y_3}(T + 1) = 0$ .

Finally, we can calculate the derivatives that we are really interested in. For the weights, which appear only in equation 96, but which affect  $Y_2$  at *all* times  $t$ , the chain rule for ordered derivatives yields:

$$F_{W_1} = \sum_{t=1}^T F_{Y_2}(t) * Y_1(t) \quad (108)$$

$$F_{W_2} = \sum_{t=3}^T F_{Y_2}(t) * Y_1(t-2). \quad (109)$$

Likewise, for the inputs  $Y_1(t)$ , we get:

$$F_{Y_1}(t) = F_{Y_2}(t) * W_1 + F_{Y_2}(t+2) * W_2 + \underline{F_{Y_3}(t+1) * 3} \quad (110)$$

Equations 106 through 110, together, form a well-ordered backwards system, which we can easily program directly into a computer program or even into a spreadsheet (as in [4]). When debugging complex systems, it often pays to identify select out simple test problems (after testing the big system), and to implement them in a spreadsheet. The computer program to calculate the derivatives would look something like:

```
Initialize  $F_{W_1}$ ,  $F_{W_2}$ , and quantities for  $t > T$  to zero
DO FOR  $t = T$  to  $t = 1$  by  $-1$ :
  Invoke equation 106
  Invoke equation 107
  Invoke equation 110
  Set  $F_{W_1}$  to  $F_{W_1} + F_{Y_2}(t) * Y_1(t)$ 
  Set  $F_{W_2}$  to  $F_{W_2} + F_{Y_2}(t+2) * Y_1(t)$ 
End
```

For very large, nonmodular forwards systems (a hundred equations or more), it is sometimes inconvenient to have to search through the entire forwards system all over again for each variable. In that case, it may be convenient to write " $F_{varN} =$ " at the front of a blank line, " $F_{varN} - 1 =$ " under that (using variable names instead of numbers), until you have a list, going over several pages, in reverse order. Then you can work your way through each occurrence of each variable as you come to it, adding the appropriate term to the right feedback equation. This kind of inverse search would also allow more efficient automation of this procedure. If you have any doubts about the details, you can use the example above as a test.

### 10.6.3. How to Construct Dual or Adjoint Subroutines or Functions

Complex systems for neurocontrol or system identification can be very difficult to keep track of if one attempts to write out every equation in the entire system as one big mass of equations. The resulting computer implementation would be even harder to maintain, understand, improve, and debug. On the other hand, the use of modular design—both in research and in implementation—can make these systems quite simple.

A vital part of these designs is the ability to backpropagate *through* different modules. For example, we may have a module that calculates:

$$z_{m+1} = f_{m+1}(z_1, \dots, z_m) \quad (111)$$

$$z_{N-1} = f_{N-1}(z_1, \dots, z_{N-2})$$

but does *not* take the final step of calculating *TARGET*,  $z_N$ . We assume that the final *TARGET* will be calculated somewhere else. The module here will simply output  $z_{m+1}$  through  $z_{N-1}$ . In order to backpropagate through this module, we must assume that some *other* module will provide the quantities:

$$FO_{-z_i} \triangleq \frac{\partial TARGET}{\partial z_i} \quad m+1 \leq i \leq N-1. \quad (112)$$

(In actuality, these derivatives may themselves be calculated by backpropagating through some other module.) We must program a *dual* module that *inputs*  $FO_{-z_i}$  and implements:

$$F_{-z_i} = FO_{-z_i} + \sum_{j=i+1}^{N-1} F_{-z_j} * \frac{\partial z_j}{\partial z_i}. \quad (113)$$

To work out what these equations are for any particular ordered system, we can use exactly the same procedures we used in section 10.6.2. Once again, this entire procedure could easily be automated. This would permit control theorists (and modelers of all kinds) to take advantage of backpropagation and the designs of neurocontrol without being restricted to the case of ANNs.

## 10.7. GRADIENT AND HESSIAN CALCULATIONS FOR SECTION 10.3

The rapid, parallel implementation of the methods of section 10.3 would depend on a rapid calculation of the gradient and (if possible) the Hessian of  $H$ , as defined in equation 20, holding the Lagrange multipliers constant. These calculations can be derived by using the chain rule for ordered derivatives. To do this (following section 10.6), we must first specify the forwards system used to calculate the target,  $H$ . For simplicity, we will use  $k$  as the “time” index (representing  $t+k$ ), use  $\phi$  instead of  $F_2$ , and use  $p$  instead of  $pH$ . We will use the notation  $s'(x)$  for the derivatives of the sigmoid function. (Actually,  $s'(x) = s(x)(1-s(x))$ .)

The forwards system can be written as the following, for  $k = 0$  to 30:

$$v_j^-(k) = W_{j0} + W_{j1}p(k-1) + W_{j2}p(k-2) + W_{j3}p(k-3) \quad (114)$$

$$+ W_{j4}\phi(k) + W_{j5}\phi(k-1) + W_{j6}\phi(k-2) + W_{j7}\phi(k-3)$$

$$x_j^-(k) = s(v_j^-(k)) \quad (115)$$

$$v_1^+(k) = W_{10}^+ + \sum_{j=1}^h W_{1j}^+ x_j^-(k) \quad (116)$$

$$p(k) = bias + s(v_1^+(k)) \quad (117)$$

$$U(k) = U(k-1) + (p(k) - p^*)^2 + \lambda_k^{(1)}(p(k) - p_{\min}) \\ + \lambda_k^{(2)}(p_{\max} - p(k)) + \lambda_k^{(3)}(\phi(k) - \phi_{\min}) + \lambda_k^{(4)}(\phi_{\max} - \phi(k))$$

where  $U(-1) = 0$  and the target ( $H$ ) is  $U(30)$ .

A direct application of the chain rule for ordered derivatives (section 10.6) yields:

$$F_{-p}(k) = 2(p(k) - p^*) + \lambda_k^{(1)} - \lambda_k^{(2)} \quad (118)$$

$$+ \sum_{j=1}^h (F_{-v_j^-(k+1)} * W_{j1} + F_{-v_j^-(k+2)} * W_{j2} + F_{-v_j^-(k+3)} * W_{j3})$$

$$F_{-v_1^+(k)} = F_{-p}(k) * s'(v_1^+(k)) \quad (119)$$

$$F_{-x_j^-(k)} = F_{-v_1^+(k)} * W_{1j}^+ \quad (120)$$

$$F_{-v_1^-(k)} = F_{-x_j^-(k)} * s'(v_1^-(k)) \quad (121)$$

$$F_{-\phi}(k) = \sum_{j=1}^h (F_{-v_j^-(k)} * W_{j4} + F_{-v_j^-(k+1)} * W_{j5} \\ + F_{-v_j^-(k+2)} * W_{j6} + F_{-v_j^-(k+3)} * W_{j7}) \quad (122)$$

where we again adopt the convention that  $F_{-anything}(k)$  equals zero for  $k > 30$ . Equation 122 yields the desired gradient (i.e., it yields  $F_{-\phi}(k)$  for  $k$  from 0 to 30). As in section 10.6, the backwards equations must be invoked backwards in time (i.e., from  $k = 30$  back to  $k = 0$ ). To explicitly implement the additional requirement (dynamic equation) that  $\phi(k+7) = \phi(7)$ , we could simply modify equation 122 to:

$$F_{-}\phi(k) = c(k) * F_{-}\phi(k+1) + \sum_j (F_{-}v_j^-(k) * W_{j4} + F_{-}v_j^-(k+1) * W_{j5} + F_{-}v_j^-(k+2) * W_{j6} + F_{-}v_j^-(k+3) * W_{j7}), \quad (123)$$

where we define  $c(k)$  as 1 for  $k > 6$  and 0 for  $k < 7$ . Again, it is easy to generalize this procedure for more complex models.

To calculate the Hessian is substantially trickier. We cannot calculate the entire Hessian in *one* pass; however, we *can* calculate the second derivatives with respect to  $\phi(i)$  and  $\phi(k)$ , for a *particular*  $i$  and *all*  $k$ , in one pass. By repeating this process for all  $i$ , we can get the Hessian at minimum cost.

To generate these equations, we must treat the *entire* system, from equation 114 to equation 122, as an *ordered* system for calculating the target  $F_{-}\phi(i)$ . We must “forget” that quantities beginning with “ $F_{-}$ ” have anything to do with derivatives. To avoid confusion, we must use “ $G_{-}$ ” to represent ordered derivatives of  $F_{-}\phi(i)$  with respect to other quantities in that ordered system. We must keep in mind that the order of calculation is from  $k = 0$  to  $k = 30$  for the forwards block and  $k = T$  to  $k = i$  for the backwards block. Working back through that system, we arrive at the following equations (which have not been implemented or tested as yet):

$$G_{-}F_{-}\phi(k) = 1 \text{ (for } k = i) \text{ or } 0 \text{ (otherwise)} \quad (124)$$

$$G_{-}F_{-}v_j^-(k) = G_{-}F_{-}\phi(k) * W_{j4} + G_{-}F_{-}\phi(k-1) * W_{j5} + G_{-}F_{-}\phi(k-2) * W_{j6} + G_{-}F_{-}\phi(k-3) * W_{j7} \quad (125)$$

$$G_{-}F_{-}x_j^-(k) = G_{-}F_{-}v_j^-(k) * s'(v_j^-(k)) \quad (126)$$

$$G_{-}F_{-}v_i^+(k) = \sum_j G_{-}F_{-}x_j^-(k) * W_{ij} \quad (127)$$

$$G_{-}F_{-}p(k) = G_{-}F_{-}v_i^+(k) * s'(v_i^+(k)) \quad (128)$$

$$G_{-}F_{-}U(k) = 0$$

$$G_{-}p(k) = 2 * G_{-}F_{-}p(k) + \sum_{j=1}^h (G_{-}v_j^-(k+1) * W_{j1} + G_{-}v_j^-(k+2) * W_{j2} + G_{-}v_j^-(k+3) * W_{j3}) \quad (129)$$

$$G_{-}v_i^+(k) = G_{-}F_{-}v_i^+(k) * F_{-}p(k) * s''(v_i^+(k)) + G_{-}p(k) * s'(v_i^+(k)) \quad (130)$$

$$G_{-}x_j^-(k) = G_{-}v_i^+(k) * W_{ij}^* \quad (131)$$

$$G_{-}v_j^-(k) = G_{-}F_{-}v_j^-(k) * F_{-}x_j^-(k) * s''(v_j^-(k)) + G_{-}x_j^-(k) * s'(v_j^-(k)) \quad (132)$$

$$G_{\varphi}(k) = \sum_{j=1}^h (G_{v_j}(k) * W_{j4} + G_{v_j}(k+1) * W_{j5} + G_{v_j}(k+2) * W_{j6} + G_{v_j}(k+3) * W_{j7}). \quad (133)$$

Of course, equation 133 yields the relevant second derivatives. Once again, multiple passes—for different  $t$ —are needed in this case, because we are looking for an entire matrix of second derivatives. In coding these equations, one first loops forward from  $k = 1$  to  $k = T$  in invoking equations 124 through 128 as a block; then one invokes through 129 through 133 for  $k = T$  back to  $k = 0$ . At least for large models, these calculations for the Hessian are an order of  $N$  less expensive than those originally proposed by Saint-Donat et al. [1].

In other applications, where we only need one vector based on second derivatives, we can usually get by with a single pass through the system, if we define the appropriate scalar *TARGET* as a function of first derivatives. An example is given in [21].

## 10.8. IMPLEMENTATION OF MODELS AND METHODS IN SECTION 10.4

Section 10.4 has already given the concepts behind these methods and their implementation; however, it left out the details of the derivative calculations, which will be given here. (For the real-time version of this, see section 10.4.6.3.)

When debugging these programs, the user is urged to print out (or store) the following three items, at a minimum, in each iteration: (1) the change in error from iteration  $k$  to iteration  $k + 1$ ; (2) the *prediction* of that change, based on the dot product of the weight change with the gradient calculated in iteration  $k$ ; (3) the prediction of the same quantity, based on the gradient calculated in iteration  $k + 1$ . Normally, items 1 and 2 are negative, and 1 lies between 2 and 3. If 1 is not between 2 and 3 (except for a few atypical cases like the first iteration), suspect a bad gradient; test the gradient as described in section 10.6. If 1 is positive, your learning rate is too high; you need to rescale your variables or improve your learning-rate algorithm. If 1 is always negative, but you are making little progress, you need a faster learning rate or you need to consider some of the tricks cited in this book to speed things up.

### 10.8.1. The Compromise and Pure Robust Methods for Feedforward ANNs or NARX Models

For the sake of brevity and generality, we will use the “modular” approach of section 10.6.3. We will assume a model of order 3, just to make the equations more readable; the generalization to arbitrary order is trivial.

Our first job is to spell out the calculations used to obtain the loss function for a given set of parameters; from equations 63 through 65, we may write:

$$\hat{Y}(t) = f(Y(t-1), Y(t-2), Y(t-3), u(t), W) \quad (134)$$



$$\hat{Y}_i(t) = (1 - w_i)\hat{Y}_i(t) + w_i Y_i(t) \quad i = 1, n \quad (135)$$

$$E = \sum_i L(\hat{Y}(t), Y(t), w). \quad (136)$$

To implement the pure robust method, simply set  $w_i = 0$ , use the *same* derivative equations as given below, but skip the equations used only to adapt  $w_i$ . In equation 134, the parameters or weights of the model— $W$ —are described as a vector, just for convenience; there is no assumption that the number of parameters is in any way related to the number of variables in the model. The vector  $u(t)$  represents all exogenous variables *used* to predict  $Y(t)$ , without regard to their actual source in space and time. The loss function  $L$  could be any of the various loss functions described in section 10.4.

Using the procedures of section 10.6, we may immediately deduce:

$$F_{\hat{Y}_i(t)} = \frac{\partial}{\partial \hat{Y}_i(t)} L(\hat{Y}(t), Y(t), W) \quad (137)$$

$$\begin{aligned} F_{\hat{Y}}(t) = & F1\_f(\hat{Y}(t), \hat{Y}(t-1), \hat{Y}(t-2), u(t+1), W, F_{\hat{Y}}(t+1)) \\ & + F2\_f(\hat{Y}(t+1), \hat{Y}(t), \hat{Y}(t-1), u(t+2), W, F_{\hat{Y}}(t+2)) \\ & + F3\_f(\hat{Y}(t+2), \hat{Y}(t+1), \hat{Y}(t), u(t+3), W, F_{\hat{Y}}(t+3)) \end{aligned} \quad (138)$$

$$F_W = \sum_i FW\_f(\hat{Y}(t), \hat{Y}(t-1), \hat{Y}(t-2), u(t), W, F_{\hat{Y}}(t)) \quad (139)$$

$$F_{w_i} = \sum_i F_{\hat{Y}_i(t)} * (Y_i(t) - \hat{Y}_i(t)). \quad (140)$$

The required derivatives are calculated by equations 139 and 140. In equations 138 and 139, we assume that you have coded up the dual subroutine,  $F\_f$ , for the function  $f$ . We have used the abbreviation " $F1\_$ " (instead of  $F_{Y(t-1)}f$ ) for the subroutine which calculates derivatives with respect to the *first* vector argument of  $f$ , and likewise for  $F2$  and  $F3$ , just to avoid possible confusion about time lags.

In practice, it is easier to calculate the sums in equations 138 and 139 as *running* sums (as with the weight derivatives), in computer programming. This yields code that begins by initializing many things to zero, followed by the main loop:

```
DO FOR t = T to t = 1 by -1:
  Invoke equation 137;
  FOR k = 1 to k = 3:  $F_{\hat{Y}}(t-k) = F_{\hat{Y}}(t-k) + Fk\_f(..., F_{\hat{Y}}(t));$ 
   $F_W = F_W + FW\_f(..., F_{\hat{Y}}(t))$ 
  FOR i = 1 to i = n:  $F_{w_i} = F_{w_i} + F_{\hat{Y}_i}(t+k) * (Y_i(t+k) - \hat{Y}_i(t+k));$ 
End
```

The next to last instruction uses " $t + k$ " instead of " $t$ ," to make sure that  $F\_Y_i$  has been completely added up; therefore, after the core loop, one needs a quick loop for  $k = 1$  to  $k = 3$  to add in the final components of  $F\_w_i$ . The cost of running code like this will be far less if you actually use a *single* subroutine, with multiple outputs, to provide all the derivatives based on  $F\_Y(t)$ .

There is one more detail of relevance to computer programmers. The dual subroutine usually requires knowledge of the *intermediate* variables of the original model  $f$ . This can be handled either by *storing* all the intermediate information on the initial forwards pass, or by *recalculating* them from the inputs of the model (which need to be stored in any case). The tradeoff here is very application-dependent.

### 10.8.2. Classical Adaptation of Time-lagged Recurrent Networks (TLRN)

For a classical TLRN with classical maximum likelihood training, the calculation of error is based on equations 53, 54, 55, 56, and 22, in that order. If we apply the chain rule for ordered derivatives (section 10.6) to that ordered system, we immediately arrive at the following equations to calculate the derivatives we need:

$$F\_Y_i(t) = \hat{Y}_i(t) - Y_i(t) \quad 1 \leq i \leq n \quad (141)$$

$$F\_x_i(t) = F\_Y_{i-n}(t) + \sum_{j=i+1}^{N+n} F\_v_j(t) W_{ji} \quad (142)$$

$$+ \sum_{j=m+1}^{N+n} F\_v_j(t+1) W'_{ji} \quad m < i \leq N+n$$

$$F\_v_i(t) = F\_x_i(t) * s'(v_i(t)) \quad m < i \leq N+n \quad (143)$$

$$F\_W_{ij} = \sum_t F\_v_i(t) * x_j(t) \quad (144)$$

$$F\_W'_{ij} = \sum_t F\_v_i(t) * x_j(t-1) \quad (145)$$

where we adopt the conventions that  $F\_Y_k$  is zero for  $k < 1$  and that  $F\_anything(t)$  is zero for  $t > T$ . The generalization for longer time lags is trivial [22].

For a sticky TLRN with classical training, the calculation of error is based on equations 53, 57, 55, 56, and 22, in that order. The derivative equations are:

$$F\_Y_i(t) = \hat{Y}_i(t) - Y_i(t) \quad (146)$$

$$F\_x_i(t) = F\_Y_{i-n}(t) + \sum_{j=i+1}^{N+n} F\_v_j(t) * W_{ji} \quad (147)$$

$$F_{-}v_i(t) = F_{-}x_i(t) * s'(v_i(t)) + \sum_{j=m+1}^{N+n} F_{-}v_j(t+1) * W_{ji}' \quad (148)$$

$$F_{-}W_{ij} = \sum_i F_{-}v_i(t) * x_j(t) \quad (149)$$

$$F_{-}W_{ij}' = \sum_i F_{-}v_i(t) * v_j(t-1) \quad (150)$$

The case of a mixed system, with a combination of classical, sticky, and extreme sticky neurons can again be handled by a straightforward use of the procedures in section 10.6.

## 10.9. THE COMPROMISE METHOD FOR RECURRENT MODELS (ANN OR OTHER)

As in section 10.8.1, we will use a modular approach here.

The notation here is somewhat tricky, since we need to consider a forwards model which outputs *two* vectors—a prediction,  $\hat{Y}(t)$ , and a filtered state vector,  $R(t)$ . In section 10.8.2, we assumed that the vector  $R(t)$  was made up of  $x_{m+1}(t)$  through  $x_{N+n}(t)$  or  $v_{m+1}(t)$  through  $v_{N+n}(t)$ , though in most applications we restrict the recurrence to a subset of the neurons. For simplicity here, I will assume a model that is only second-order in the predicted variables and first-order in the recurrent variables. We may write the forwards system used to evaluate error as:

$$\hat{Y}(t) = f_1(\bar{Y}(t-1), \bar{Y}(t-2), R(t-1), u(t), W) \quad (151)$$

$$R(t) = f_2(\bar{Y}(t-1), \bar{Y}(t-2), R(t-1), u(t), W) \quad (152)$$

$$\bar{Y}_i(t) = (1 - w_i)\hat{Y}_i(t) + w_i Y_i(t) \quad (153)$$

$$E = \sum_i L(\hat{Y}(t), Y(t), W). \quad (154)$$

If we apply the chain rule for ordered derivatives, as we did in section 10.8.1, we end up with equation 137 followed by:

$$F_{-}\bar{Y}(t) = F1_{-}f(\bar{Y}(t), \bar{Y}(t-1), R(t), u(t+1), W, F_{-}\hat{Y}(t+1), F_{-}R(t+1)) \\ + F2_{-}f(\bar{Y}(t+1), \bar{Y}(t), R(t+1), u(t+2), W, F_{-}\hat{Y}(t+2), F_{-}R(t+2)) \quad (155)$$

$$F_{-}R(t) = F3_{-}f(\bar{Y}(t), \bar{Y}(t-1), R(t), u(t+1), W, F_{-}\hat{Y}(t+1), F_{-}R(t+1)) \quad (156)$$

$$F_{-}W = \sum_i F_{-}W_{-}f(\bar{Y}(t-1), \bar{Y}(t-2), R(t-1), u(t), W, F_{-}\hat{Y}(t), F_{-}R(t)) \quad (157)$$

followed by equation 140. The key thing about these equations is that there was really only *one* forwards model  $f$  with *two outputs*; if we have  $F_{-}$  derivatives available for both of these outputs, we can use a *single* dual subroutine to calculate the derivatives with respect to all four of the relevant vector arguments of  $f$ .

Clearly, the main difficulty in implementing equations 155 through 10.8.24 is the need to program the dual subroutine, based on the methods of section 10.6. Perhaps an example might help. Suppose that the recurrent system under discussion is a classical TLRN. In that case, the "model" may take the form:

$$x_i(t) = Y_i(t-1) \quad 1 \leq i \leq n \quad (158)$$

$$x_{i+n}(t) = Y_i(t-2) \quad 1 \leq i \leq n \quad (159)$$

$$x_{i+2n}(t) = R_i(t-1) \quad 1 \leq i \leq r \quad (160)$$

$$x_{i+2n+r}(t) = u_i(t) \quad 1 \leq i \leq m-2n-r \quad (161)$$

$$v_i(t) = \sum_{j=1}^{i-1} W_{ij} x_j(t) \quad m < i \leq N+n \quad (162)$$

$$x_i(t) = s(v_i(t)) \quad m < i \leq N+n \quad (163)$$

$$\hat{Y}_i(t) = x_{N+i}(t) \quad 1 \leq i \leq N+n \quad (164)$$

$$R_i(t) = x_{N+r+i}(t) \quad 1 \leq i \leq r. \quad (165)$$

Equations 158 through 165 can be coded into *one subroutine* that produces two outputs— $Y(t)$  and  $R(t)$ . Applying the methods of section 10.6, we arrive at the equations for the dual subroutine:

$$F_{-}x_i(t) = \sum_{j>i} F_{-}v_j(t) * W_{ji} + F_{-}\hat{Y}_{i-N}(t) + F_{-}R_{i-N+r}(t) \quad (166)$$

$$F_{-}v_i(t) = F_{-}x_i(t) * s'(v_i(t)) \quad (167)$$

$$F_{-}W_{ij}(t) = F_{-}v_i(t) * x_j(t) \quad (168)$$

$$F_{-}R_i(t-1) = F_{-}x_{i+2n}(t) \quad (169)$$

$$F_{-}\hat{Y}_i(t-2) = F_{-}x_{i+n}(t) \quad (F2_{-}) \quad (170)$$

$$F_{-}\hat{Y}_i(t-1) = F_{-}x_i(t) \quad (F1_{-}). \quad (171)$$

Equations 166 through 171 can be coded into a *single* dual subroutine that yields all four  $F_-$  arrays needed in equations 155 through 157. In equation 166, as in earlier examples, we treat  $F_{-\hat{Y}_i}$  and  $F_{-R_i}$  as zero for subscripts that are out of bounds.

Once again, equations 151 through 154 can be used with *any* model  $f$ , so long as you know how to program the dual subroutine for  $f$ . For example,  $f$  could be a simultaneous-recurrent neural network or some other recurrent structure (such as a fuzzy inference structure or a finite element code or a simultaneous model from econometrics); to program a dual subroutine for such an  $f$ , see Chapter 13. The insertion of such an  $f$  into this system allows one to build a single modeling system that combines together the advantages of simultaneous recurrence, time-lagged recurrence, and robust estimation; once again, we believe that the higher centers of the brain rely on a similar combination (along with real-time learning) [66,67].

## 10.10 REFERENCES

- [1] J. Saint-Donat, N. Bhat, and T. J. McAvoy, Neural net based model predictive control. *Int. J. Control*, in press.
- [2] N. Bhat and T. McAvoy, Information retrieval from a stripped backpropagation network. In *AIChE Proceedings*, 1990.
- [3] P. J. Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*, Ph.D. thesis, Division of Engineering and Applied Physics, Harvard University, Cambridge, MA, 1974.
- [4] P. J. Werbos, Maximizing long-term gas industry profits in two minutes in Lotus using neural network methods. *IEEE Trans. Systems, Man & Cybernetics*, March/April 1989.
- [5] M. J. Piovoso et al., Neural network process control. In *Proceedings: Analysis of Neural Networks Applications Conference*. ACM Order #604911. New York: ACM, 1991.
- [6] T. J. McAvoy et al., Interpreting biosensor data via backpropagation. *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, IEEE, New York, 1989.
- [7] N. Bhat and T. McAvoy, Use of neural nets for dynamic modeling and control of chemical process systems. *1989 American Control Conference; Computers and Chemical Engineering*, in press.
- [8] T. McAvoy, Y. Arkun, and E. Zafiriou, eds., *Model-Based Process Control: Proceedings of the 1988 IFAC Workshop*, Pergamon press, Oxford, U.K., 1989.
- [9] C. Gingrich, K. Kuespert, and T. McAvoy, Modeling human operators using neural networks. In *Proceedings of Instrument Society of America Conference*, October 1990.
- [10] D. Kuespat and T. McAvoy, Neural network behavior analysis of human process operators. In *AIChE Proceedings*, 1991.

- [11] G. Birky and T. J. McAvoy, A neural net to learn the design of distillation controls, preprints of IFAC DYCORN Symposium, pp.147–153, Maastricht, Netherlands, August 1989.
- [12] S. Naidu, E. Zafirov, and T. McAvoy, Application of neural networks with detection of sensor failure during the operation of a control system. *IEEE Control Systems*, 10:49–55, April 1990.
- [13] W. Cary et al., Chemometric analysis of multisensor arrays. *Sensors and Actuators*, 1:223–234, 1986.
- [14] T. Tanaka et al., A trouble forecasting system by a multi-neural network on a continuous casting process of steel production. *Artificial Neural Networks, Vol. 1*, T. Kohonen et al., eds., North-Holland Amsterdam, 1991.
- [15] P. J. Werbos, Neurocontrol and fuzzy logic. *International Journal of Approximate Reasoning*, February 1992. An earlier, less complete version appeared in *Proceedings of the Second Joint Technology Workshop on Neural Networks and Fuzzy Logic (1990)*, Vol. II, NASA Conference Publication 10061. See also P. Werbos, Elastic fuzzy logic: a better fit to Neurocontrol. *I:zuka-92 Proceedings*, Japan, 1992.
- [16] P. J. Werbos, Making diagnostics work in the real world. *Handbook of Neural Computing Applications*, A. Maren, ed., , pp.337–338, Academic Press, New York, 1990.
- [17] E. Blum, Approximation theory and feedforward networks, *Neural Networks*, 4, 1991.
- [18] G. Cybenko, Approximation by superpositions of sigmoidal functions. *Mathematics of Controls, Signals and Systems*, 2:303–314, 1988.
- [19] E. Sontag, Feedback stabilization using two-hidden-layer nets. SYCON-90-11. Rutgers University Center for Systems and Control, New Brunswick, NJ, October 1990.
- [20] P. J. Werbos, Applications of advances in nonlinear sensitivity analysis. *Systems Modeling and Optimization: Proceedings of the 1981 IFIP Conference*, R. Drenick and F. Kozin, eds., Springer-Verlag, New York, 1982.
- [21] P. J. Werbos, Backpropagation: Past and future. *ICNN Proceedings*. IEEE, New York, 1988. A transcript of the talk, with slides, is somewhat more readable and is available from the author.
- [22] P. J. Werbos, Backpropagation through time: What it is and how to do it. *Proceedings of the IEEE*, October 1990.
- [23] D. Rumelhart, G. Hinton, and R. Williams, *Parallel Distributed Processing*, MIT Press, Cambridge, MA, 1986.

- [24] D. F. Shanno, Recent advances in numerical techniques for large-scale optimization. *Neural Networks for Control*, W. Miller, R. Sutton, and P. J. Werbos, eds., MIT Press, Cambridge, MA, 1990.
- [25] See also Chapter 13.
- [26] E. R. Panier and A. L. Tits, *On Feasibility, Descent and Superlinear Convergence in Inequality Constrained Optimization*. Systems Research Center, University of Maryland, College Park, MD, 1989.
- [27] J. Zhou and A. L. Tits, *User's Guide for FSQP*, Systems Research Center, University of Maryland, College Park, MD, 1989.
- [28] P. J. Werbos, A menu of designs for reinforcement learning. In W. Miller, R. Sutton, and P. J. Werbos, *op. cit.* [24].
- [29] T. Su, T. J. McAvoy, and P. J. Werbos, Identification of chemical processes via parallel training of neural networks. Submitted to *IECE*, 1991.
- [30] C. F. Mosteller and R. E. Rourke, *Sturdy Statistics: Nonparametric and Order Statistics*, Addison-Wesley, Reading, MA, 1973.
- [31] L. Ljung, *System Identification, Theory for the User*, Prentice-Hall, Englewood Cliffs, NJ, 1987.
- [32] P. Young, Parameter estimation for continuous-time models: a summary. *Automatica*, 17:(1), 1981.
- [33] K. S. Narendra and K. Parthasarathy, Identification and control of dynamic systems using neural networks. *IEEE Trans. Neural Networks*, 1, 1990.
- [34] P. J. Werbos, Learning how the world works: Specifications for predictive networks in robots and brains. In *Proceedings of the SMC Conference*. IEEE, New York, 1987.
- [35] P. J. Werbos, Neurocontrol and related techniques. In A. Maren, *op. cit.* [16].
- [36] P. J. Werbos, Advanced forecasting methods for global crisis warning and models of intelligence. *General Systems Yearbook*, 1977.
- [37] P. J. Werbos and J. Titus, An empirical test of new forecasting methods derived from a theory of intelligence: the prediction of conflict in Latin America. *IEEE Trans. Systems, Man & Cybernetics*, September 1978.
- [38] M. Kawato, Computational schemes and neural network models for formation and control of multijoint arm trajectory. In W. Miller, R. Sutton, and P. J. Werbos, *op.cit.* [24].

- [39] B. Widrow and M. E. Hoff, Adaptive switching circuits. *WESCON Convention Record, IV*, pp. 96–104. Reprinted in *WESCON/89*.
- [40] B. A. Pearlmutter, Dynamic recurrent neural networks. Technical report CMU-CS-90-196. Carnegie-Mellon University, Pittsburgh, PA, 1990.
- [41] N. Baba, A new approach for finding the global minimum of error function of neural networks. *Neural Networks*, 5:367–373, 1989.
- [42] F. J. Solis and J. B. Wets, Minimization by random search techniques. *Mathematics of Operations Research*, 6:19–30, 1981.
- [43] M. J. Willis et al., On artificial neural networks in process engineering. *IEEE, Part D*, 1991.
- [44] H. J. Bremermann and R. W. Anderson, An alternative to backpropagation: a simple rule for synaptic modification for neural net training and theory. Department of Mathematics, University of California at Berkeley, 1989.
- [45] K. M. Pedersen, M. Kummel, and H. Soeberg, Monitoring and control of biological removal of phosphorus and nitrogen by flow injection analyzers in a municipal pilot scale wastewater treatment plant. Submitted to *Analytical Chim. Acta*, 1990.
- [46] J. H. Gary and G. E. Handwerk, *Petroleum Refining: Technology and Economics*. Marcel Dekker, New York, 1975.
- [47] R. J. Williams and J. Peng, An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural Computation*, 4, 1990.
- [48] H. Koivisto, *Minimum Prediction Error Neural Controller*, Department of Electrical Engineering, Tampere University of Technology, Tampere, Finland, 1990.
- [49] D. Nguyen, *Applications on Neural Networks in Adaptive Control*, Ph.D. thesis. Stanford University, Palo Alto, CA, June 1991.
- [50] G. Box and G. Jenkins, *Time-Series Analysis: Forecasting and Control*, Holden-Day, San Francisco, CA, 1970.
- [51] I. J. Leontaritis and S. A. Billings, Input-output parametric models for nonlinear systems. *Int. J. Control*, 41:303–344, 1985.
- [52] A. E. Bryson and Y. C. Ho, *Applied Optimal Control*, Ginn and Co., Waltham, MA, 1969.
- [53] A. Gelb, ed., *Applied Optimal Estimation*. MIT Press, Cambridge, MA, 1974.



- [54] P. J. Werbos, Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1:(4) (submitted August 1987).
- [55] P. J. Werbos, Energy and population: Transitional issues and eventual limits. *The Elephants in the Volkswagen: Facing the Tough Questions About Our Crowded Nation*, L. Grant, ed., W. H. Freeman, New York, 1992.
- [56] P. J. Werbos, Econometric techniques: Theory versus practice. *Energy: The International Journal*, March/April 1990. (Also forthcoming in a Pergamon book edited by Weyant and Kuczmowski.)
- [57] R. J. Solomonoff, A formal theory of inductive inference. *Information and Control*, March/June 1964.
- [58] N. Tishby, E. Levin, and S. Solla, Consistent inference of probabilities in layered networks: Predictions and generalization. In *IJCNN Proceedings, Vol. II*, IEEE, New York, 1989.
- [59] P. Papantoni-Kazakos and D. Kazakos, Fundamental neural structures, operations and asymptotic performance criteria in decentralized binary hypothesis testing. In *Proceedings of the IEEE Conference on Neural Networks for Ocean Engineering*, IEEE No. 91CH3064-3. IEEE, New York, 1991.
- [60] A. J. Dempster, *J. American Statistical Assoc.*, 72:(357), March 1977.
- [61] P. J. Werbos and J. Titus, *Predicting Conflict in Latin America: The First Full Test of Werbos's Robust Method*. Final report of DARPA project funded through ONR contract No. N00014-75-C-0846. This document was submitted in January 1978 to DARPA, but never entered into the defense documentation system.
- [62] H. Hjalmarsson and L. Ljung, How to estimate model uncertainty in the case of undermodelling. *1990 American Control Conference*, IEEE, New York, 1990.
- [63] G. C. Goodwin and R. L. Payne, *Dynamic System Identification: Experiment Design and Data Analysis*, Academic Press, New York, 1977.
- [64] K. Niida, J. Tani, and I. Koshijima, Application of neural networks to rule extraction for operator data. *Computer and Chemical Engineering*, submitted in 1990.
- [65] A. R. Barron, Predicted square error: a criterion for automatic model selection. *Self-Organizing Methods in Modeling*, S. J. Farlow, ed.
- [66] P. J. Werbos, The cytoskeleton: Why it may be crucial to human learning and to neurocontrol. *Nanobiology*, 1:(1), 1991 (a new journal from Carfax).
- [67] P. J. Werbos, Neurocontrol, biology and the mind. In *IEEE Proceedings SMC*, IEEE, New York, 1991.

# APPROXIMATE DYNAMIC PROGRAMMING FOR REAL-TIME CONTROL AND NEURAL MODELING

*Paul J. Werbos*

*NSF Program Director for Neuroengineering  
Co-director for Emerging Technologies Initiation<sup>1</sup>*

## 13.1. INTRODUCTION

The title of this chapter may seem a bit provocative, but it describes rather precisely what my goals are here: to describe how certain control designs, based on an approximation of dynamic programming, could someday reproduce the key capabilities of biological brains—the ability to learn in real time, the ability to cope with noise, the ability to control many actuators in parallel, and the ability to “plan” over time in a complex way. These are ambitious goals, but the brain itself is an existent proof that they are possible. The new material in this chapter has been discussed with prominent neurophysiologists in the past few months, and remarkable parallels with the human brain have been identified [1,2]; however, this chapter will mention only a few highlights of the more recent discussions as they bear upon the engineering.

Chapter 3 has already shown that the neurocontrol community has developed two general families of designs capable of planning or optimization to some degree, over time—the backpropagation of utility and adaptive critics. Of the two, only adaptive critics show real promise of achieving the *combination* of capabilities mentioned in the previous paragraph. This chapter will begin to fill in several crucial gaps between the long-term promise mentioned above and the existing methods described in Chapters 3 and 10:

---

<sup>1</sup> The views expressed here are those of the author, not those of NSF.

1. Section 2 will provide detailed information on alternative methods to adapt a Critic network (i.e., to approximate a solution of the Bellman equation of dynamic programming).
2. Section 3 will provide preliminary information on the strengths and weaknesses of the alternative methods, based on linear examples (which could also be used in debugging and in establishing links with other methods).
3. Section 4 will discuss how neurocontrol systems can fulfill functions like hierarchical planning, chunking, etc., which are major concerns of the artificial intelligence (AI) community.
4. Section 5 will describe an Error Critic design that adapts time-lagged recurrent networks in real time, *without* the cost-scaling problems of the usual forward perturbation methods.
5. The methods of Chapter 10 adapt neural networks to yield good *predictions* of a plant or environment. To make full use of stochastic methods, one needs to adapt networks that represent correct *probability distributions*. Section 6 will present a Stochastic Encoder/Decoder/Predictor architecture that can do so.

The issue of exploration is also important to these designs, but other chapters of this book will have more to say about it.

Like Chapter 3, this chapter will try to present the basic algorithms in such a way that *any* reasonable functional form can be used—linear or nonlinear, neural or nonneural. Therefore, it will have to assume a full understanding of Chapter 3 and of Appendix B of Chapter 10. For example, it will frequently use dual subroutines or dual functions. If  $Y=f(\text{arguments})$  is a differentiable function, and  $x$  is one of the arguments of  $f$ , then we may define the dual function of  $f$  with respect to  $x$  as:

$$F_{f_x}(\text{arguments}, F_Y) \triangleq \text{the vector } \sum_i F_{Y_i} * \frac{\partial f_i}{\partial x_j}(\text{arguments}). \quad (1)$$

This notation for differentiation takes getting used to. However, if  $f$  represents a sparsely connected nonlinear system (like a neural net), then we can calculate all these dual functions in a single sweep, at a much lower cost than the brute force matrix multiplication suggested by equation 1; thus, dual functions are crucial to the computational efficiency of our algorithms.

Likewise, this chapter will assume some understanding of supervised learning. A real-time supervised learning system may be defined as a set of admissible vector-valued functions,  $f(W, \text{other arguments})$ , and a procedure for updating  $W$  when presented with a set of values for the other arguments and a desired value (or “target”) for  $f$ . For the sake of generality, this chapter will not restrict itself to any one choice of supervised learning system, but it will usually require functions  $f$  that are continuous everywhere and differentiable almost everywhere.

There will be *no simulation studies* reported in this chapter. Simulation studies are an important tool in the nonlinear case, but a thorough mathematical understanding of the linear case (or even the finite-state case) is often the best place to start with a new algorithm. Narendra’s stability proof for the nonlinear case in this book is far more satisfying than the simple linear checks used here; however, Narendra’s work was *possible* only because of decades of painstaking work nailing down the properties of the linear versions of his methods, followed by many nonlinear simulations guided by the resulting intuition. For the control problems and algorithms addressed in this chapter, we are still at an earlier point in that cycle.

## 13.2. METHODS FOR ADAPTING CRITIC NETWORKS

### 2.1. Introduction and Notation

Chapter 3 mentioned a number of ways to adapt Critic networks. This section will provide details of implementation for four of those methods—heuristic dynamic programming (HDP) and dual heuristic programming (DHP), both in their original forms and in their action-dependent variations (ADHDP and ADDHP). It will not discuss certain other important methods, such as GDHP [3,4] or methods based on instrumental conditioning theories [5], because they are two steps beyond the present state of the art in published real-world applications; however, it will briefly discuss the method of temporal differences (TD) of Barto, Sutton, and Anderson [6].

I will assume that the controller and the environment interact as follows, at each time  $t$ :

1. An estimate of the state vector,  $R(t)$ , becomes available.
2. The controller performs its various calculations and invokes the Action network  $A$  to calculate the actions  $y(t) = A(R(t))$ . Total utility,  $U(R(t), u(t))$  is also calculated.
3. The action  $u(t)$  is transmitted to the environment.

(Some authors would prefer to assume that  $U$  is *observed*, rather than *calculated*, but the assumption here is more general and more realistic [1,3].) Portions of this section will assume the availability of a Model network:

$$\hat{R}(t+1) = f(R(t), u(t)), \quad (2)$$

but I will not discuss the adaptation of the Model network or its use in estimating the state vector  $R$ , because this is discussed elsewhere (see Chapter 10). For the same reason, I will not discuss the adaptation of the Action network here (see Chapter 3). Likewise, there is no reason here to specifically refer to the weights or parameters of those networks.

In all four methods, the Critic network can be adapted using *any* supervised learning method. Thus, the Critic network could be an MLP, a CMAC, or a Grossberg network. Our four algorithms are not alternatives to basic backpropagation or associative memory methods in adapting the *weights* of such networks. Rather, they are *procedures* for how to set up the *inputs* and *targets* of a network. Like direct inverse control (see Chapter 3), they are *generalized* procedures that can be used with *any* specific supervised learning method. All four algorithms are fully time-forwards, real-time methods as presented here.

### 2.2. Mathematical Background

This subsection is not necessary to implementing the algorithms but may be helpful in understanding them.

HDP is based on an attempt to approximate Howard's form of the Bellman equation, given as equation 29 of Chapter 3. For simplicity, I will assume problems such that we can assume  $U_0 = 0$ . (If not, see [3].) ADHDP is based on equation 31 of Chapter 3.

DHP is based on differentiating the Bellman equation. Before performing the differentiation, we have to decide how to handle  $\mathbf{u}(t)$ . One way is simply to define the *function*  $\mathbf{u}(\mathbf{R}(t))$  as that function of  $\mathbf{R}$  which, for every  $\mathbf{R}$ , maximizes the right-hand side of the Bellman equation. With that definition (for the case  $r = 0$ ), the Bellman equation becomes:

$$J(\mathbf{R}(t)) = U(\mathbf{R}(t), \mathbf{u}(\mathbf{R}(t))) + \langle J(\mathbf{R}(t+1)) \rangle - U_0. \quad (3)$$

where we must also consider how  $\mathbf{R}(t+1)$  depends on  $\mathbf{R}(t)$  and  $\mathbf{u}(\mathbf{R}(t))$ . Differentiating, and applying the chain rule, we get:

$$\begin{aligned} \lambda_i(\mathbf{R}(t)) &\triangleq \frac{\partial J(\mathbf{R}(t))}{\partial R_i(t)} = \frac{\partial}{\partial R_i(t)} U(\mathbf{R}(t), \mathbf{u}(\mathbf{R}(t))) + \langle \frac{\partial J(\mathbf{R}(t+1))}{\partial R_i(t)} \rangle \\ &= \frac{\partial U(\mathbf{R}(t), \mathbf{u}(t))}{\partial R_i(t)} + \sum_j \frac{\partial U(\mathbf{R}, \mathbf{u})}{\partial u_j} \cdot \frac{\partial u_j(\mathbf{R}(t))}{\partial R_i(t)} \\ &\quad + \sum_j \langle \frac{\partial J(\mathbf{R}(t+1))}{\partial R_j(t+1)} \cdot \frac{\partial R_j(t+1)}{\partial R_i(t)} \rangle \\ &\quad + \sum_{j,k} \langle \frac{\partial J(\mathbf{R}(t+1))}{\partial R_j(t+1)} \cdot \frac{\partial r_j(t+1)}{\partial u_k(t)} \cdot \frac{\partial u_k(t)}{\partial R_i(t)} \rangle. \end{aligned} \quad (4)$$

Strictly speaking, this calculation would have been more rigorous if we had used the chain rule for ordered derivatives given in Chapter 10; however, the content of this equation should be clear intuitively. Backpropagation “through” the Model network and the Action network, based on their dual subroutines, is simply an efficient way to carry out these chain-rule calculations without having to store and multiply large matrices.

In a similar vein, equation 31 of Chapter 3 translates into:

$$J'(\mathbf{R}(t), \mathbf{u}(t)) = U(\mathbf{R}(t), \mathbf{u}(t)) + \langle J'(\mathbf{R}(t+1), \mathbf{u}(\mathbf{R}(t+1))) \rangle - U_0 \quad (5)$$

which, when differentiated, yields:

$$\begin{aligned} \lambda_i^{(R)}(\mathbf{R}(t), \mathbf{u}(t)) &\triangleq \frac{\partial}{\partial R_i(t)} J'(\mathbf{R}(t), \mathbf{u}(t)) = \frac{\partial}{\partial R_i(t)} U(\mathbf{R}(t), \mathbf{u}(t)) \\ &\quad + \sum_j \left( \frac{\partial J'}{\partial R_j(t+1)} + \sum_k \frac{\partial J'}{\partial u_k(t+1)} \cdot \frac{\partial u_k(t+1)}{\partial R_j(t+1)} \right) \cdot \frac{\partial R_j(t+1)}{\partial R_i(t)} \end{aligned} \quad (6)$$

and a similar equation for  $\lambda^{(u)}$ . Note that our *definition* of  $\lambda^{(R)}$  in equation 6 requires us to differentiate the left-hand side of equation 5 as if  $\mathbf{u}(t)$  were a constant vector; consistency requires us to do likewise with the right-hand side in order to calculate  $\lambda^{(R)}$ . (If we had treated  $\mathbf{u}(t)$  as a function of  $\mathbf{R}$  when differentiating  $J'$ , then  $\lambda^{(R)}$  would have been just the same as the  $\lambda$  of DHP!)

In theory, even with DHP, we might have adapted a  $\lambda^{(u)}$  network, whose targets would simply be the derivatives of  $J(t+1)$  propagated back to  $u(t)$ ; however, since those derivatives are available *directly* to the Action network (as per Figure 2.7 of Chapter 3), there is no real need for such a network. Still, the calculations involved in running *one* network (the  $\lambda^{(u)}$  network) might take less time than all the steps shown in that figure; therefore, it is conceivable that such a network might have some use in lower-level Action networks that need to react very quickly.

### 2.3. Implementation of HDP, TD Methods, and Comments

HDP is a procedure for adapting a network or function,  $\hat{J}(R(t), W)$ , which attempts to approximate the function  $J(R(t))$ . Figure 13.1 illustrates the basic idea. The steps in HDP are as follows, starting from any actual or simulated value for  $R(t)$ :

1. Obtain and store  $R(t)$ .
2. Calculate  $u(t) = A(R(t))$ .
3. Obtain  $R(t+1)$ , either by waiting until  $t+1$  or by predicting  $R(t+1) = f(R(t), u(t))$ .
4. Calculate:

$$J^*(t) = U(R(t), u(t)) + \hat{J}(R(t+1), W)/(1+r) \quad (7)$$

5. Update  $W$  in  $\hat{J}(R(t), W)$  based on inputs  $R(t)$  and target  $J^*(t)$ .

Step 5 is where we use *any* real-time supervised learning method. There are a variety of ways to implement this five-step procedure, ranging from a purely on-line approach (where  $R(t)$  and  $R(t+1)$  are never simulated or predicted) to a “dreaming” approach; all are statistically consistent, *if* we assume that forecasting errors are normal and independent. (Section 6 will show how to relax that assumption.) The issue of how best to simulate  $R(t)$  when dreaming is an outstanding research issue,

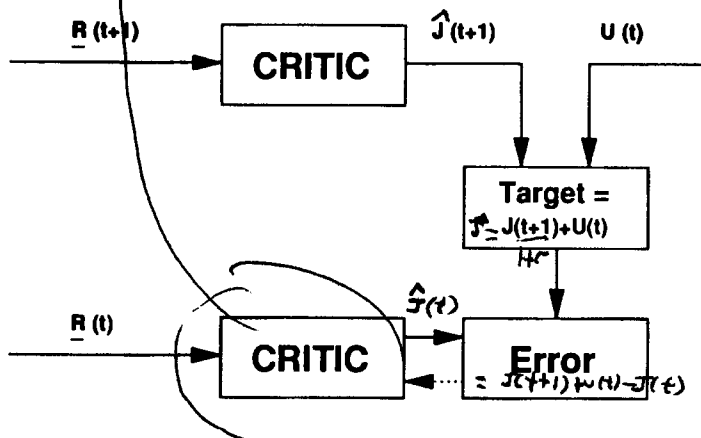


Figure 13.1 HDP as a way to adapt a Critic.

closely related to the issues of exploration and search. Because HDP is based on dynamic programming, one might expect that optimal dreaming would involve something like a backwards sweep, starting out near goal states, terminal states, or the like.

It may be possible to accelerate HDP by modifying step 5, to account for the changes in the target itself that result from changing  $W$ ; however, the obvious way of doing this (adapting  $\hat{J}(t) - \hat{J}(t+1)$  to match  $U$ , in effect) failed the consistency test we will discuss in section 3 [8].

HDP was originally developed with the thought of applying it to biological organisms or to long-term decision problems, where the true horizon of our concern is essentially infinite [7]. For that reason, I focused on the case where  $r$  (the factor we use to discount the future) is zero. The original temporal difference (TD) method of Barto, Sutton, and Anderson [6] adapted a Critic in a situation where there was a terminal time  $T$ , and the utility payoffs all occurred at time  $T$ . They adapted a table look-up kind of Critic, which was updated to match the target  $\hat{J}(t+1)/(1+r)$  for the case  $t < T$ , and to the target  $U$ , for the case  $t = T$ . In the special case where  $U = 0$  for  $t < T$  and  $J(T+1) = 0$ , it is obvious that equation 8 yields the targets they used.

A few researchers have argued that ANNs should be asynchronous systems, without clock pulses and the like, in order to reflect the distributed, asynchronous nature of the brain. The design above violates that idea. However, many well-known biologists, like Llinas, have studied clock pulses and synchronization systems in the brain in very great empirical detail. For example, it is well known that the limbic lobes of the brain (which appear to serve as a Critic network [1,4]) operate at a cycle time (theta rhythm) about twice the normal cycle time (alpha rhythm) of the cerebral cortex, which appears to calculate the short-term memory or state vector  $R(t)$  [14]. Llinas, in conversation, has stated that these cycle times are "unbelievably constant" over time in individual animals. If good clocks are difficult to achieve in organic brains, then their pervasive existence suggests very strongly that they are critical to brain-like capabilities.

## 2.4. Implementation of DHP

DHP is a procedure for adapting a Critic network or function,  $\hat{\lambda}(R(t))$ , which attempts to approximate the function  $\lambda(R(t))$  defined in equation 4. DHP, like HDP, can use *any* real-time supervised learning method to adapt the Critic. There is no need to use backpropagation in the supervised learning itself; therefore, DHP is not affected by the issue of convergence speed in basic backpropagation. Nevertheless, the procedure for calculating the *target* vector,  $\lambda^*$ , does use backpropagation in a generalized sense; more precisely, it does use dual subroutines to backpropagate derivatives through the Model network and the Action network, as shown in Figure 13.2.

DHP entails the following steps, starting from any value of  $R(t)$ :

1. Obtain  $R(t)$ ,  $u(t)$  and  $R(t+1)$  as was done with HDP.
2. Calculate:

$$\hat{\lambda}(t+1) = \hat{\lambda}(R(t+1), W) \quad (8)$$

$$F_{-u}(t) = F_{-U_u}(R(t), u(t)) + F_{-f_u}(R(t), u(t), \hat{\lambda}(t+1)) \quad (9)$$

$$\sum^* \hat{\lambda}^*(t) = F_{-f_R}(R(t), u(t), \hat{\lambda}(t+1)) + F_{-U_R}(R(t), u(t)) + F_{-A_R}(R(t), F_{-u}(t)). \quad (10)$$

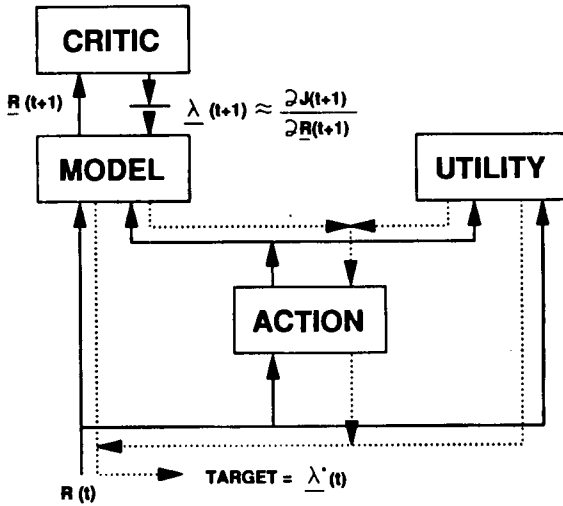


Figure 13.2 Calculating the targets  $\lambda^*$  in DHP.

3. Update  $W$  in  $\hat{J}(R(t), W)$  based on inputs  $R(t)$  and target vector  $\lambda^*(t)$ .

In these equations, note that  $\lambda(t+1)$  is input into the dual subroutine *in place of*  $F_R(t+1)$ . Notice how equations 9 and 10 are equivalent to equation 4 above, using our quick definition of a dual function in equation 1. In some earlier descriptions of DHP [3], I left out the third term by analogy to the backpropagation of utility; however, the results in section 3 below suggest that this was an error.

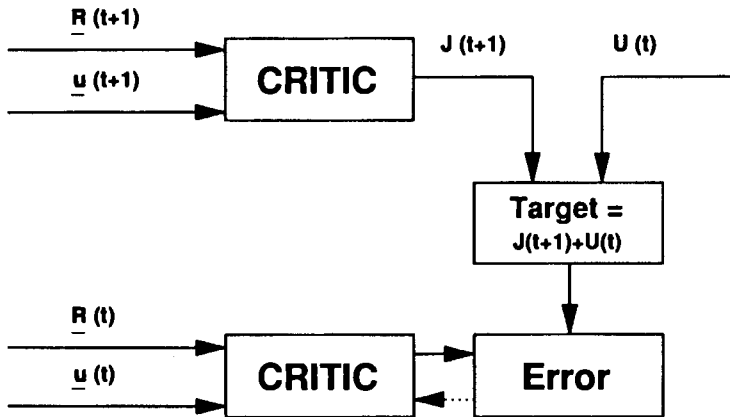
## 2.5. Implementation of ADHDP (“Q-learning”)

ADHDP adapts a Critic network,  $\hat{J}(R(t), u(t), W)$ , which attempts to approximate  $J'$  as defined in equation 31 of Chapter 3. Most implementations of ADHDP so far have followed that equation quite directly. Instead of using an Action network to generate  $u(t+1)$ , they generate *all possible* action vectors  $u(t+1)$ , calculate the resulting  $\hat{J}(t+1)$ , and then pick the  $u(t+1)$  which maximizes  $\hat{J}(t+1)$ . These calculations are used to generate the *target* that the Critic is adapted to. This nonneural approach has some obvious limitations:

- When the action vector  $u$  is complex, it is not practical to simply enumerate all its possible values.
- The procedure requires a good Model network or excellent prior information to tell us what  $\hat{J}(t+1)$  will be, for each choice for  $u(t+1)$ . (We certainly cannot *try* every  $u(t+1)$  in parallel in the actual plant!)

In addition, it leads to instability in tasks like broom-balancing, for technical reasons discussed by Watkins. For the sake of exploration, we sometimes take actions  $u(t+1)$  that are *not* optimal [9], but these are *not* used in calculating  $\hat{J}(t+1)$  when adapting the Critic network.





**Figure 13.3** ADHDP as a way to adapt an action-dependent Critic.

So far, there have only been two true neural net implementations of ADHDP, implementations in which both the Critic and the Action network are neural networks [10,11]. The method for adapting the Critic is shown in Figure 13.3. It is nearly identical to HDP.

Starting from any  $R(t)$ , the steps are:

1. Obtain  $R(t)$ ,  $u(t)$ , and  $R(t+1)$  exactly as with HDP.
2. Calculate:

$$J^*(t) = U(R(t), u(t)) + \hat{J}'(R(t+1), A(R(t+1), W)/(1+r). \quad (11)$$

3. Update  $W$  in  $\hat{J}'(R(t), u(t), W)$  based on inputs  $R(t)$  and  $u(t)$  and target  $J^*(t)$ .

## 2.6. Implementation of ADDHP

ADDHP adapts a Critic network with *two* outputs— $\lambda^{(R)}(R(t), u(t), W)$  and  $\lambda^{(u)}(R(t), u(t), W)$ —which try to approximate the two gradients of  $J'(R(t), u(t))$ . In other words, the “output vector” of the Critic network is the concatenation of two vectors. The procedure is very much like DHP, except that the targets are calculated in a different way, as shown in Figure 13.4.

Starting from any  $R(t)$ , the steps are:

1. Obtain  $R(t)$ ,  $u(t)$ , and  $R(t+1)$  exactly as in HDP.
2. Calculate  $u(t+1) = A(R(t+1))$ .
3. Calculate:

$$F\_R(t+1) = \hat{\lambda}^{(R)}(R(t+1), u(t+1), W) + F\_A_R(R(t+1), \hat{\lambda}^{(u)}(R(t+1), u(t+1), W)) \quad (12)$$

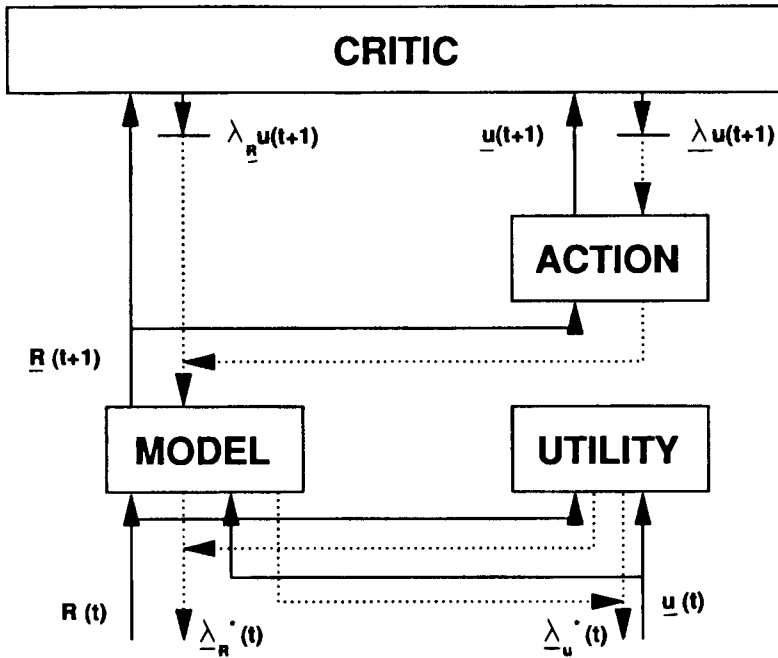


Figure 13.4 Calculating the targets  $\lambda_{\underline{R}}^*$  and  $\lambda_{\underline{u}}^*$  in ADDHP.

$$\lambda_{\underline{R}}^*(t) = F_{f_R}(R(t), u(t), F_{\underline{R}}(t+1)) + F_{U_R}(R(t), u(t)) \quad (13)$$

$$\lambda_{\underline{u}}^*(t) = F_{f_u}(R(t), u(t), F_{\underline{R}}(t+1)) + F_{U_u}(R(t), u(t)) \quad (14)$$

4. *Update  $W$  in the Critic based on inputs  $R(t)$  and  $u(t)$  and targets  $\lambda_{\underline{R}}^*(t)$  and  $\lambda_{\underline{u}}^*(t)$ .*  
Note the comparison between step 3 and equation 6 above.

### 13.3. PERFORMANCE TRADEOFFS IN LINEAR EXAMPLES

#### 3.1. Summary

This section will perform a preliminary analysis of the strengths and weaknesses of the Critic designs above, based on exact solutions in the linear case. This analysis was stimulated in part by discussions with Guy Lukes (of [9]), David White, Michael Jordan, and Andy Barto regarding their experience in simulating some of these systems. The primary conclusions are:

- ADHDP, when fully implemented as a network adaptation scheme, has problems with what Narendra [12] has called “persistence of excitation.” Near an optimal solution, it loses its ability

to learn efficiently. In the work of Sofge and White [11], this problem was limited by a combination of persistent process noise and the use of time derivatives (which magnify noise), but it still was noticeable.

- ADDHP and DHP have much less of a problem with the persistence of excitation. The Critic networks in ADDHP and DHP do not have to be MLPs and do not have to be adapted by basic backpropagation, any more than the Critics in HDP do; however, backpropagation must be used in setting up the *targets* that the Critic adapts *to*. (This is still a real-time design.) In setting up the targets, it is important to propagate derivatives through the Model network *and* the Action network; this is not like backpropagating utility, where the Action-net term is often optional. This supersedes earlier discussions of DHP, which were incomplete [3].
- The advantages of DHP over HDP can be understood somewhat analytically. They hinge on the fact that a Model network usually needs fewer inputs per output than a J-style Critic does, and on the fact that changes in an Action network do not require changes in one's Model network. Again, however, hybrid designs might have benefits in handling type 3 unexpected events, as discussed in Chapter 3.

In many applications, initial experiments with DHP should probably be based on backpropagating through the actual simulation model to be used in testing (using the methods of Chapter 10, Appendix B), instead of adapting a neural net Model network; the latter may be essential, eventually, for true real-time adaptation, but it may help to start out by understanding how the controller operates in the absence of system identification problems.

Problems also arise (beyond the scope of this chapter) in exploring new regions with HDP. Updating  $\hat{J}(t)$  to match  $\hat{J}(t+1) + U(t)$  can be problematic, when the overall level of  $J$  in a certain region is totally unknown. Since DHP deals with the *gradient* of  $J$ , instead of the overall level of  $J$ , it should behave differently in this situation.

### 3.2. A Linear-Quadratic Example

To understand these methods further, consider the following simple linear-quadratic control problem. Before evaluating the methods themselves, we must work out the correct values of various functions in this problem, so that we have something to check the methods against.

Suppose that the vector of observables,  $\mathbf{x}(t)$ , is the same as the state vector of the plant; in other words, assume that everything is fully observable. Suppose that we are trying to maximize a measure of utility,  $U(t)$ , summed from the present time to the infinite future, and defined by:

$$U(t) = -\mathbf{x}(t)^T \mathbf{Q} \mathbf{x}(t). \quad (15)$$

Suppose that the plant is governed by:

$$\mathbf{x}(t+1) = \mathbf{P}\mathbf{x}(t) + \mathbf{R}u(t) + \mathbf{e}(t). \quad (16)$$

where:

$$\langle \mathbf{e}(t)\mathbf{e}(t)^T \rangle = \mathbf{E}. \quad (17)$$

Suppose that our Action network is:

$$u(t) = Ax(t). \quad (18)$$

For any *fixed* value of the matrix  $A$ , we can calculate the  $J$  function *conditional* upon that  $A$  (following Howard's approach [13]). As a preliminary step to doing this, we define:

$$P' = P + RA, \quad (19)$$

and observe that:

$$x(t+1) = P'x(t) + e(t). \quad (20)$$

Let us define  $M$  as the matrix that solves the following equation:

$$M = P'MP'^T - Q. \quad (21)$$

(This equation can be solved explicitly if we define  $M$  as a vector in  $N^2$ -dimensional space, but the resulting expression for  $M$  is not useful for our purposes here.) I claim that Howard's form of the Bellman equation (with  $r = 0$  and  $A$  now *fixed*) is satisfied by:

$$J(x) = x^T M x. \quad (22)$$

To prove this, we simply substitute this into Howard's equation (3) and verify that it is satisfied. For the left-hand side of the equation, we get:

$$J(x(t)) = x(t)^T M x(t). \quad (23)$$

For the right-hand side, we get:

$$\begin{aligned} U(x) + \langle J(x(t+1)) \rangle &= -x^T Q x + \langle (P'x(t) + e(t))^T M (P'x(t) + e(t)) \rangle \\ &= -x^T Q x + x(t)^T P'^T M P' x(t) + \langle e(t)^T M e(t) \rangle. \end{aligned}$$

(In this last calculation, I exploit the fact that  $e$  was assumed to be random, so that its cross-covariance with  $x$  is zero.) Because  $M$  satisfied equation 21, and because the distribution of  $e$  is a constant with respect to  $x$ , this tells us that:

$$U(x) + \langle J(x(t+1)) \rangle = x^T(t) M x(t) + U_0, \quad (24)$$

where  $U_0$  is a constant (here zero). Comparing equations 22 and 24, we can see that the Bellman equation is indeed satisfied. Howard has proven [13] that we converge to an optimal strategy of action if we alternately calculate the  $J$  function for the *current* strategy of action, *modify* the strategy of action so as to maximize that  $J$ , recalculate  $J$  for the new strategy of action, and so on.

Following equation 31 of Chapter 3, it is easy to see that the true value of  $J(x(t), u(t))$  conditional upon the present action strategy  $A$  is:

$$J = -x(t)^T Q x(t) + (P x(t) + R u(t))^T M (P x(t) + R u(t)). \quad (25)$$

### 3.3. Persistence of Excitation

The need for persistent excitation is one of the classic problems of adaptive control [12], which can be dealt with but never fully exorcised.

In extreme form, let us imagine a control problem where we are supposed to force a plant to stay very close to a desired operating point,  $x_0$ . Suppose that the level of noise in the plant is extremely low. In this case, a good controller will learn very quickly to make sure that there is very little change or variance in the state of the plant or in the action vector itself over time. However, from a statistical point of view, it is *precisely* the variance (or “excitation”) of the plant that allows us to understand how the plant works; as that variance decreases, our understanding may deteriorate, leading to a deterioration in control, until the variance grows large enough to allow a slow return to adaptive control. This kind of effect led to many surprising, counterintuitive failures of sensible-looking control schemes in the 1960s and 1970s.

How does this apply to adaptive critic systems?

From equations 19 through 25, we can see that the weights in any form of Critic network should *change* as the strategy of action,  $A$ , changes. The network must therefore be capable of “unlearning” the past. This makes it especially important that there be excitation or variance in more recent time periods.

In the case of ADHDP, let us consider a simple linear problem in which the optimal action turns out to be  $u = -x$  (i.e.,  $A = -I$ ). In the early stages of adaptation, the  $A$  matrix will be incorrect; therefore, the Critic must learn to “forget” those stages, and pay more attention to information near the optimum, as it approaches the optimum. Near the optimum, however, it is trying to predict some target using inputs— $u$  and  $x$ —that are totally correlated with each other. It is well known from elementary statistics [15] that this kind of situation leads to huge inaccuracy in estimating weights; this, in turn, implies inaccuracy in adapting the Critic. With least-squares estimation techniques, like basic backpropagation and regression, the estimates of the weights in the Critic will still be consistent estimators but the random errors in these estimates will grow inversely with the variance of  $u + x$  [15]. Other forms of supervised learning would tend to do even worse here, because they do not correct so precisely for correlations between input variables. Random exploration could help a little here, by reducing the correlation between  $u$  and  $x$ .

With DHP, the situation would be radically different, because we only use *one* input vector— $x$ —to predict the vector of targets. Thus, the correlation between  $u$  and  $x$  is not a problem. Even with ADDHP, we use a *Model* network to consider explicitly how *changes* in  $u$  would *change* the results at time  $t + 1$ ; therefore, even if  $u$  should be correlated with  $x$ , the procedure will still be informed (“excited”) about *alternative* possibilities for  $u$ . With DHP and ADDHP, one must still worry about maintaining the validity of the *Model* network as the system experiences different regions of control space; however, the *Model* network—unlike the Critic network—should not have to change as the Action network changes. In other words, Howard’s procedure [13] tells us to adapt the Critic  $J$  by

throwing out data based on earlier, suboptimal action strategies  $A$ , *even if* the plant is unchanged; however, for any given plant, data from optimal *and* suboptimal action strategies can be used without difficulty in system identification. Therefore, the excitation problems should be far less in the case of DHP. With DHP, there is no strong mechanism present to increase Model errors over time, near a stable equilibrium, because there is no strong need to overwrite earlier experience in that situation.

These conclusions are reinforced by considering how DHP and ADDHP would work for the simple linear problem given above, *when* the correct model (equation 16) is known.

### 3.4. DHP in the Linear-quadratic Example

Let us check to make sure that DHP works properly, for a *given* value of  $A$  and a *known* model, in the simple linear-quadratic example above. More precisely, let us check to see—if the Critic is initially correct—that it will *stay* correct after an adaptation step. (This check still leaves open a lot of other questions for future research.)

From equation 30, the correct value of  $J(x)$  is  $x^T M x$ . The correct value for  $\lambda(x)$  is simply the gradient of this. Thus, we start from:

$$\lambda(t+1) = 2Mx(t+1).$$

Our task is to calculate the targets for  $\hat{\lambda}(t)$  as they would be generated by DHP, and then check them against the correct values. To do this, we must carry out the calculations implied by equations 9 and 10 for this case. To calculate the first term on the right-hand side of equation 10, we propagate  $\lambda(t+1)$  through the model (equation 16) back to  $x(t)$ , which yields an expected value of:

$$\langle P^T(2Mx(t+1)) \rangle = 2P^T M(P+RA)x(t).$$

For the second term, we simply use the gradient of  $U(x(t))$ :

$$-2Qx(t).$$

For the third term, we propagate  $\lambda(t+1)$  through the model (equation 16) back to  $u(t)$ , and then through the Action network, which yields an expected value of:

$$\langle A^T(R^T(2Mx(t+1))) \rangle = 2A^T R^T M(P+RA)x(t). \quad (26)$$

Adding all three terms together, we find that the expected value of our target vector will be:

$$\langle \lambda^*(t) \rangle = 2(P+RA)^T M(P+RA)x(t) - 2Qx(t), \quad (27)$$

which is easily seen to equal  $2Mx(t)$ —the correct value—if we exploit equation 21. The smaller the learning rate, the more we can be sure of averaging out the effect of noise and tracking the expectation value; here, as elsewhere in neural nets, there is a tradeoff between speed and accuracy, which is best resolved by using adaptive learning rates, as in Chapter 3.

Some earlier accounts of DHP did not mention the third term, the term in equation 26. If this term were not present, the left-hand "RA" in equation 27 would disappear, and the matrix multiplying  $x(t)$  would no longer be symmetric, unless  $RA$  were zero. Thus, the Critic would be grossly incorrect.  $RA$  is zero only under extreme and uninteresting conditions—conditions where the control actions have no effect at all on the system to be controlled, or where the best control strategy is always to do nothing. Therefore, neglect of the third term leads to the wrong result, almost always.

This situation is quite different from what we get in backpropagating utility. With the backpropagation of utility, one can always derive an optimal *schedule* of actions, starting from fixed initial conditions at time  $t = 1$  and going through to a terminal time  $t = T$ . This is possible because we assume the total absence of noise when using that method. The optimal schedule of actions can be found simply by calculating the derivative of utility with respect to action *at each time*, without allowing for any Action network at all. With a powerful enough Action network, one could presumably reproduce this same optimal schedule of actions as a function of state input variables. Thus, there should be no need to propagate derivatives through the Action network in that case—unless the Action network is so constrained that it lacks the ability to reproduce an optimal or near-optimal strategy. Again, this is quite different from DHP.

Notice as well that the calculations above do not become problematic for  $A$  near an optimum. The persistence of excitation problems we saw with ADHDP do not appear here, *so long as*  $P$  and  $R$  are known.

### 3.5. ADDHP in the Example

As with DHP, we will assume that the Critic is correct at time  $t + 1$ , and verify that the expected values of the targets are correct.

Differentiating equation 25 with respect to  $u$ , at time  $t + 1$ , we start with:

$$\begin{aligned} \langle \lambda_u(t+1) \rangle &= \langle 2R^T M(Px(t+1) + Ru(t+1)) \rangle \\ &= 2R^T M(P + RA) \langle x(t+1) \rangle = 2R^T M P' \langle x(t+1) \rangle \end{aligned} \quad (28)$$

and likewise with  $R$ :

$$\langle \lambda_R(t+1) \rangle = -2Q \langle x(t+1) \rangle + 2P^T M P' \langle x(t+1) \rangle. \quad (29)$$

Following Figure 13.4, we first propagate  $\lambda^{(u)}$  through the Action network, and add the result to  $\lambda^{(R)}$  to get:

$$\begin{aligned} \langle F_R(t+1) \rangle &= \lambda_R(t+1) + A^T(\lambda_u(t+1)) \\ &= -2Q \langle x(t+1) \rangle + 2(P + RA)^T M P' \langle x(t+1) \rangle \\ &= -2Q \langle x(t+1) \rangle + 2P'^T M P' \langle x(t+1) \rangle. \end{aligned}$$

By equation 37 this is simply:

$$\langle F_R(t+1) \rangle = 2Mx(t+1) = 2M(Px(t) + Ru(t)).$$

(Notice that this complete derivative corresponds to the  $\lambda(t+1)$  of DHP!) To calculate the target for  $\lambda^{(n)}$ , we propagate *this* through the Model network back to  $u(t)$ , and add in the gradient of utility with respect to  $u(t)$  (which happens to be zero in this example); the result is:

$$\langle \lambda_u^*(t) \rangle = 2R^T M(Px(t) + Ru(t)), \quad (30)$$

which corresponds to the correct value that we get from differentiating equation 25. In a similar way, the targets for  $\lambda^{(k)}$  are also correct. As with DHP, propagation of derivatives through the Action network was crucial to getting the right answer.

### 3.6. DHP versus HDP

A previous paper [8] described the performance of HDP in an even simpler example, where  $A$  was again fixed. This example may be written as:

$$\begin{aligned} x(t+1) &= P'x(t) + e(T) \\ U(x) &= U^T x \\ J(x) &= w^T x. \end{aligned} \quad (31)$$

(Note that  $U$  was a *vector* in that study, and that there was no need to assume positive or negative definiteness.) It showed that we would arrive at the following update for  $w$  after a long series of experiments in which  $A$  was fixed:

$$\langle w^{(n)} \rangle = P'^T w^{(n-1)} + U. \quad (32)$$

However, as with all expected values, there is noise. To get smaller and smaller errors in  $w$ , one needs exponentially increasing amounts of data [15]. One needs to work through *many* values of  $t$ . Furthermore, since equation 32 is actually the result of regressing all the  $x_i$  variables on a single dependent variable, the errors will be magnified further if there are correlations between these variables. With DHP in the same problem, we would use a very simple Critic network:

$$\lambda(x) = w.$$

Using calculations like those above, we see that a *single* time cycle is enough to yield *exactly*:

$$\lambda^* = P'^T w + U. \quad (33)$$

Once the correct model is known, then, DHP can be immensely faster. Of course, it takes time to learn the model. Therefore, a realistic tradeoff would compare the errors implied by equation 32 versus those in estimating a model. The standard errors [15] are determined in part by the degree of collinearity and the number of independent variables; one would expect both factors to be much larger in equation 32—where  $J$  depends on *all* variables in the system and represents *all* direct and indirect connections through time—than in estimating a model from time  $t$  to  $t+1$ , because the patterns of *direct* causal links will usually tend to be sparse.



If it is difficult to find a good functional form for the Model network—despite all the theorems that say that MLPs can approximate *any* functional form—HDP may still have some advantages in terms of robustness; this, in turn, suggests that a hybrid approach might be the ultimate way to go. For now, however, the advantages of equation 33 over equation 32 are very compelling for large problems, and (as discussed in Chapter 3) there is reason to believe this is a general conclusion.

### 13.4. ADAPTIVE CRITICS AND ARTIFICIAL INTELLIGENCE

During two of the workshops that stimulated this book, there was considerable discussion of some very basic issues, including a key question: To what extent can practical, realistic, adaptive critic designs reproduce those aspects of *intelligence* that receive the most attention from the AI community? Adaptive critics do have a close relation to the “evaluation functions” or “static position evaluators” of classical AI.

Some of this discussion was of enormous scientific importance; however, the more abstract analysis of linguistic reasoning [1] and planning [16] has been published elsewhere, and will not be repeated here in detail. From an engineering applications point of view, this analysis has led to the following informed conjectures which merit further investigation:

1. *The designs of section 2 will indeed have limited capability if the boxes in the figures are all filled in with feedforward networks.* Such designs are likely to work very well in ordinary but difficult control problems, like complex problems of controlling a nonlinear aircraft or managing a chemical plant. However, they are likely to perform poorly in problems like robot navigation *through* a crowded workspace, or problems like intercepting multiple missiles in an optimal fashion. The problem is that even *after learning*, no feedforward system—not even a human—can be expected to glance at a *novel* complex scene and instantly see the path through the maze; even *after learning*, humans seem to implement something like a relaxation algorithm that requires them to keep looking at the scene until a pattern emerges.

The obvious solution here is to use a *simultaneous-recurrent* network, as described in Chapter 3, as the Critic network. Minsky showed long ago [17] that the problem of detecting *connectivity* in a visual scene can be solved parsimoniously by a recurrent network, but not by feedforward MLPs; presumably, the search for a connected path through a cluttered workspace imposes similar computational requirements. Also, there are reasons to expect better performance from such networks in regard to generalization and parsimony (see Chapter 10), temporal chunking [16], etc. In writing Chapter 3 and section 2, I have deliberately tried to specify the algorithms so that you can plug in this option directly, without additional information.

There would be certain difficulties, however, in making this option practical. Suppose, for example, that it takes 100 iterations through the underlying feedforward network ( $f$  in equation 18 of Chapter 3) to achieve converged output for the simultaneous-recurrent network  $F$ . Clearly, if we need frequent control actions out of the overall controller (e.g., 1,000 hertz operation), we would need very fast operation in the feedforward network (100,000 hertz in this example!).

Surprisingly, there is evidence that the human brain actually does this in its higher centers; Walter Freeman has stated that the recurrent inner loop of the hippocampus runs at 400 hertz, even though the overall system runs at something more like 4 hertz [1].

Stability may be a serious issue with systems like this. Indeed, occasional instability may be an unavoidable price of very high intelligence. It is hard to imagine these networks obeying Lipschitz conditions of the kind that Narendra's theorem requires (though Lipschitz conditions can probably be maintained, with proper care, for a wide range of feedforward nets). Therefore, it might turn out to be important to avoid using such powerful systems in applications where "Terminator 2" modes of behavior are available and overrides could be bypassed.

To take all this a step further—*either* with feedforward Critics (if timing requires their use) *or* with simultaneous-recurrent nets—one can modify DHP as follows. Instead of adapting  $\lambda(R(t), W)$ , adapt  $\hat{\lambda}(R(t), \hat{\lambda}(R(t-1)), W)$ , using the exact same procedure given in section 2. Treat the  $\lambda(t-1)$  argument as a *constant* input, and do *not* use the methods of Chapter 10 to backpropagate derivatives back to time  $t-1$ , etc. Backpropagation through time (or the equivalent) is crucial for adapting short-term memory units, but the purpose of recurrence here is *not* to add additional memory. The assumptions of dynamic programming suggest that the short-term memory should be implemented *inside* the Model network (which defines the state vector), not within the Critic network. Adding this lagged input to the Critic is like giving a human chess player more than a quarter-second, if he needs it, to evaluate or critique a complex strategic situation.

2. *In neurocontrol, complex hierarchies and distributed metasystems are usually not necessary.* Neural networks can easily implement hierarchical or distributed calculations *simply by choosing* an appropriate pattern of connectivity (i.e., by zeroing out weights  $W_{ij}$  outside a designated graph of connections). When we have *prior knowledge* that tells us a hierarchical solution to a problem will work, it does make sense to *use* that knowledge in setting up the initial state of the network; however, it also makes sense later on to give us and the network freedom to make and break connections through "pruning" (see Chapter 10) and "random" exploration of possible new connections. A *single* neural network is *itself* a distributed system, and it can be implemented as a network spanning many locations. The well-known biologist Pribram has argued that the brain is a "heterarchy," not a hierarchy.

When *combining* neural nets and other architectures, however, hierarchical designs can make more sense. For example, telerobotic control of the main arm of the space shuttle presents a severe challenge to classical control theory. Seraji developed a hierarchical control scheme that worked on a Puma robot arm to some degree, but was computationally intensive and never deployed. This past year [18], the joint controllers used by Seraji were replaced by neurocontrollers (using direct inverse control and computationally affordable), with a substantial improvement in performance, at least in simulations. The NASA program officer has authorized tests on the real shuttle arm, and hopes for a tenfold increase in productivity compared to the present teleoperation system. Even though there is no theoretical need for hierarchies when working with neural nets, there is certainly no harm in using them to get a quick startup in applications of this sort.

3. *A simple two-level hierarchy can nevertheless be useful when the underlying optimization problem requires a long cycle time to analyze, but high frequency control is needed.* In this situation, we could build one adaptive critic system—including a Critic and a Model network, but *not necessarily* an Action network—to operate with a long cycle time,  $\theta$ , long enough to let us use simultaneous-recurrent networks. Then we could build another adaptive critic system, using a short time cycle (1), based on feedforward networks. The second system would

be “independent” of the first, except that in adapting the low-level Critic, in equation 7, we would replace  $U(R(t), u(t))$  with something like:

$$\hat{J}_-(R(t_- + \theta)) + U_-(R(t_-)) - \hat{J}_-(R(t_-)) + U_-(R(t), u(t)), \quad (34)$$

where  $t_-$  is the most recent value of  $t$  divisible by  $\theta$  (i.e., the latest clock tick for the upper system), where  $\hat{J}_+$  is the output of the upper Critic, and  $U_-$  indicates high-frequency components of utility that cannot be tracked at the upper level. (Total utility equals  $U_+ + U_-$ .) (A similar arrangement could be adapted to DHP or to an upper level of GDHP and a lower level of DHP.) The lower system would also be allowed to *input* information—to be treated like an additional set of external observables—from the upper system. In a variant of this, we could allow the upper system to have an Action network, and calculate the *total* action vector  $u(t)$  as the sum of  $u_+(t_+ + \theta)$  and  $u_-(t)$ . There is evidence that the human brain uses some such arrangement [1,2]; for example, Llinas has recently studied 8 to 10 hertz muscle tremors in humans due to the low frequency of signals from the cerebral cortex down to the olive, which appears to act as a Critic for the lower motor control system. The lower motor system is analyzed in more detail as a neurocontroller in [25].

### 13.5. ERROR CRITICS: AN APPROACH TO THE REAL-TIME ADAPTATION OF TLRNS

The problem of forecasting may be seen as a simple generalization of the problem of supervised learning. As before, we start out by observing two sets of variables over time,  $X(t)$  and  $Y(t)$ . As before, we wish to adapt a network that outputs a forecast of  $Y(t)$ , based on our knowledge of  $X(t)$ . The only difference is that we *permit* our network to remember information from earlier time periods. In any applications—such as adaptive control—this kind of memory is crucial; in fact, it gets to be absurd to think of control problems so simple that the state of the plant depends *only* on current control actions and *not at all* on its previous state!

Nevertheless, because of our emphasis on control here, I will formulate the forecasting problem in a different but equivalent way, tied directly to the needs of adaptive critic systems. Let us suppose that we are trying to forecast a vector of observed variables,  $X(t + 1)$ , as a function  $f$  of the state at time  $t$ , the action vector  $u(t)$ , and weights  $W$ . Following the procedures of Chapter 10, we will choose  $f$  to be a time-lagged recurrent network (TLRN) and allow for the possibility that  $\hat{X}(t)$  could be an input to the network as well. In the neural network field, people usually visualize recurrent nodes,  $R_i(t)$ , as being “inside” the network; however, it is easier to describe our methods in the general case by treating the output of recurrent nodes as an *output* of the network, an output that is then *input* to the network in the next time period. Putting all of this together, we are trying to minimize error in the following scheme, where  $f$  is a *single* static network implemented by a *single* subroutine, with *two* output vectors ( $f_X$  and  $f_R$ ):

$$\hat{X}(t + 1) = f_X(X(t), \hat{X}(t), R(t), u(t), W) \quad (35)$$

$$R(t+1) = f_R(X(t), \hat{X}(t), R(t), u(t), W) \quad (36)$$

$$Error = \sum_i E(t) = \sum_i L(X(t), \hat{X}(t), W). \quad (37)$$

As an example,  $f$  could be a two-stage system, consisting of an MLP which inputs  $R(t)$ ,  $u(t)$ , and  $\hat{X}(t)$ , and a simple filtering system which calculates  $\hat{X}(t)$  as a weighted average of  $X(t)$  and  $\hat{X}(t)$  (see Chapter 10). Intuitively,  $R(t)$  in this system represents memory from *before* time  $t$ . (The  $R$  in this system is slightly different from the  $R$  of earlier sections, but there is a close connection.)

There are two exact ways to adapt the system in equations 35 through 37. The usual time-forwards approach has been discussed elsewhere (see Chapters 3 and 5) and will not be discussed here. The cheaper approach, backpropagation through time (BTT), has also been discussed elsewhere (see Chapter 10) but is important as a first step in understanding the Error Critic.

Using BTT, one starts from initial guesses for the weights ( $W$  and  $W'$ ), one calculates the gradient of error across all times  $t$  with respect to the weights, and one adapts the weights in response to the gradient. The only new complication lies in how to calculate the gradient. If we ignore times 0 and  $T$  (which will not affect our real-time learning method), BTT yields the following equations for the gradient here:

$$F_{\hat{X}}(t) = F_{L_{\hat{X}}}(X(t), \hat{X}(t), W) + F_{f_{\hat{X}}}(X(t), \hat{X}(t), R(t), u(t), W, F_{\hat{X}}(t+1), F_{R}(t+1)) \quad (38)$$

$$F_{R}(t) = F_{f_R}(X(t), \hat{X}(t), R(t), u(t), W, F_{\hat{X}}(t+1), F_{R}(t+1)) \quad (39)$$

$$F_W = F_W + F_{L_W}(X(t), \hat{X}(t), W) + F_{f_W}(X(t), \hat{X}(t), R(t), u(t), W, F_{\hat{X}}(t+1), F_{R}(t+1)). \quad (40)$$

As in Chapter 10 (see Appendices B and D), we only need to call *one* dual subroutine,  $F_f$ , which yields all *three* of the outputs required here ( $F_{f_{\hat{X}}}$ ,  $F_{f_R}$ , and  $F_{f_W}$ ). Equations 38 and 39 force us to go backwards through time, since  $F_{\hat{X}}(t)$  and  $F_{R}(t)$  cannot be calculated until after  $F_{\hat{X}}(t+1)$  and  $F_{R}(t+1)$  are known.

In a previous paper [19], I have proposed that we treat the problem of adapting the recurrent hidden nodes as a *control problem*. More precisely, I proposed that we turn this problem on its head, by treating  $E(t)$  as a measure of utility and by treating  $R(t)$  as a *control* signal. To solve that control problem over time, in a *real-time* manner, I proposed the use of an adaptive critic control system [19].

Schmidhuber [20] has actually attempted this approach, using a Critic network adapted by HDP. Unfortunately, HDP has many limitations, discussed in detail above. The limitations of HDP relative to DHP are *especially severe* when there is an *exact model* already available. In this application, the forecasting network itself already provides what amounts to an exact model of how error is generated as a function of  $R(t)$ .

To apply DHP or ADDHP to this problem, one might go through severe agonizing to try to identify the appropriate state vector, and so on. Fortunately, there are *very* straightforward ways to use these methods here that do not require anything like that degree of complexity.

Let us define a Critic network,  $\lambda$ , which inputs  $X(t)$ ,  $X(t-1)$ ,  $u(t)$ ,  $u(t-1)$ , etc., and outputs two quantities ( $\lambda_R$  and  $\lambda_X$ ), which attempt to approximate:

$$\lambda_{\hat{X}}(X(t) \dots, W_{\lambda}) \approx F_{\lambda\hat{X}}(X(t), \hat{X}(t), R(t), u(t), W_{\lambda}, F_{\hat{X}}(t+1), F_{R}(t+1)) \quad (41)$$

$$\lambda_{\hat{R}}(X(t) \dots, W_{\lambda}) \approx F_{\lambda\hat{R}}(X(t), \hat{X}(t), R(t), u(t), W_{\lambda}, F_{\hat{X}}(t+1), F_{R}(t+1)). \quad (42)$$

There are several ways to implement and use this kind of Error Critic. For example, we could go through the following steps for any time  $t$ :

1. Obtain (observe)  $X(t)$ .
2. Calculate  $u(t) = A(X(t), R(t))$ .
3. Calculate:

$$\begin{aligned} \lambda_{\hat{X}}(t) &= F_{\lambda\hat{X}}(X(t), \hat{X}(t), W) + \lambda_{\hat{X}}(X(t), \dots, W_{\lambda}) \\ \lambda_{\hat{R}}(t) &= \lambda_{\hat{R}}(X(t), \dots, W_{\lambda}). \end{aligned}$$

4. Set:

$$\begin{aligned} W &= W - LR(t) * (F_{\lambda W}(X(t), \hat{X}(t), W) \\ &\quad + F_{\lambda W}(X(t-1), \hat{X}(t-1), R(t-1), u(t-1), W, \lambda_{\hat{X}}(t), \lambda_{\hat{R}}(t))). \end{aligned}$$

where  $LR(t)$  is a learning rate (that can be adapted over time, etc., as in Chapter 3).

5. Update  $W_{\lambda}$  in  $\lambda$  to inputs  $(X(t-1), X(t-2), \dots)$  and targets:

$$\begin{aligned} \lambda^* \hat{X} &= F_{\lambda\hat{X}}(X(t-1), \hat{X}(t-1), R(t-1), u(t-1), W, \lambda_{\hat{X}}(t), \lambda_{\hat{R}}(t)) \\ \lambda^* \hat{R} &= F_{\lambda\hat{R}}(X(t-1), \hat{X}(t-1), R(t-1), u(t-1), W, \lambda_{\hat{X}}(t), \lambda_{\hat{R}}(t)). \end{aligned}$$

6. Calculate and store for the next round:

$$\begin{aligned} \hat{X}(t+1) &= f_{\hat{X}}(X(t), \hat{X}(t), R(t), u(t), W) \\ R(t+1) &= f_R(X(t), \hat{X}(t), R(t), u(t), W). \end{aligned}$$

As a practical matter, of course, there are many ways to try to “pipeline” these calculations. For example, one might merge  $f$  and  $\lambda$  into a single network; however, with  $\lambda$  as an output, it is essential to have inputs for *two* consecutive time periods, and experimentation will be needed to find the most effective subset of the allowable inputs. Alternatively, one might split  $f$  up into two different networks, so that  $R(t+1)$  could be input instead of  $R(t)$  to the Action network. (This would be closer in spirit to our prior sections.) Or one could adapt the Critic only on every second time cycle. One could allow the Critic to accept input only from time  $t$ , but *then* use feedback from derivatives propagated through the Critic to affect the adaptation of  $R$  neurons used as input to the Critic. One might even try to use GDHP in this application.

In early tests of this possibility, it would be good to compare against backpropagation through time (and possibly against simple truncation, which approximates the difficult cross-time terms as zero); after all, the Error Critic is still only an approximation to BTT.

From a biological point of view, this arrangement may seem moderately complex, but the giant pyramid cells of the cerebral cortex involve a very similar kind of complexity (including an external clock pulse used to synchronize the various flows of information). The same arrangement, but without equation 37, can be used to adapt an Action network with time-lagged recurrence, something which can be useful when there is an extreme need to boost computational speed even at the cost of slower learning (as in the cerebellum [25]).

When building a complex neurocontrol system, it can sometimes be confusing to have *one* Critic network evaluating actions and another adapting recurrent neurons. Therefore, I sometimes use the notation  $\lambda_E$  or  $\lambda^{(E)}$  to represent the Error Critic.

## 13.6. HOW TO ADAPT AND USE A TRULY STOCHASTIC MODEL NETWORK

### 6.1. Introduction

In order to adapt forecasting networks, most neural network researchers now use equations similar to equations 35 through 37 in section 5. When Model networks or system identification are needed, in *any* of the five forms of neurocontrol, those researchers would simply adapt a forecasting network. In most current applications, that is probably good enough.

Nevertheless, the theory behind adaptive critics allows for the possibility of a *general* stochastic model. It is common to build a stochastic model by simply writing:

$$X_i(t+1) = \hat{X}_i(t+1) + \sigma_i e_i(t+1), \quad (43)$$

where  $e_i$  represents random noise of unit variance, and  $\sigma_i^2$  represents the variance of the error in predicting  $X_i$ . (In other words,  $\sigma_i^2$  is simply the average value of the squared error.) One can build a neural network to generate the forecasts, using the methods of section 5, and estimate  $\sigma_i$  simply by measuring the error.

Section 2 and Chapter 3 described ways of adapting Critic and Action networks based on observing the *actual* values of  $R(t+1)$ , and feeding back derivatives through the Model network. That procedure is legitimate for stochastic models built up from equation 43. It is equally legitimate to use a *hypothetical*  $R(t)$  as the starting point in the same procedure, and to *simulate*  $R(t+1)$  using equation 43, and then to treat the simulated  $R(t+1)$  as if it were real. Such a simulation required that we generate random numbers to represent the  $e_i$ .

Unfortunately, equation 43 does *not* represent a general stochastic model. It assumes that the matrix  $\langle ee^T \rangle$  is diagonal, and that the randomness in the *observed* variables always follows a normal distribution. Conventional control theory usually allows for *any* matrix  $\langle ee^T \rangle$ , although it does have problems with nonnormal distributions.

In 1977 [7], I suggested that we consider a more general model that may be written as:

$$X_i(t+1) = \sigma_i^x e_i^x(t+1) + D_i(R(t+1), \text{other information}(t)) \quad (44)$$

$$R_i(t+1) = \sigma_i^R e_i^R(t+1) + P_i(\text{information}(t)), \quad (45)$$

where  $X$  is observed and  $R$  is not.  $D$  stands for “Decoder,” and  $P$  for “Predictor.” This architecture can reproduce essentially any patterns of noise we like. Those researchers who have proven that MLPs can approximate any bounded function might want to study the ability of networks like this to reproduce arbitrary probability distributions.

The challenge here lies in how to adapt the networks  $D$  and  $P$  when  $R$  is unknown. The classical likelihood function for this problem involves integrals over all possible values of  $R$  [7]. We could approximate these integrals, in theory, by performing a Monte Carlo integration/simulation, directly based on equations 44 and 45, but this turns out to be extremely inefficient from a numerical point of view. This section will propose a new design, which should be more efficient. The reader should be warned, however, that the adaptation problem here is extremely difficult. The design below has passed some simple tests of consistency, but this is not enough to prove that it will work without revisions in the general case. When the “Predictor” network is removed, this design can also be used as a kind of unsupervised feature extractor; however, it is radically different from other forms of unsupervised learning now in use in the neural network field.

The method below is a slight variant of a method that I first proposed in 1977 [7]. That method was developed as a way of addressing the nonlinear case, based on integrals that represent the concept of “relative entropy” or “mutual information.” The variant given here is different, above all, because it does pass the basic consistency checks in the linear case.

The method below may be thought of as a nonlinear, neural-net generalization of a well-known classical statistical method: maximum likelihood factor analysis [21]. Maximum likelihood factor analysis does have difficulties when one of the noise terms ( $\sigma_i^X$ ) is infinitely small. This makes it desirable to replace the maximum likelihood approach with a more robust approach. That will be a task for future researchers.

The final part of this section will discuss how the method below ties in with classical debates on realism versus phenomenology, and on fuzzy versus probabilistic reasoning.

## 6.2. The Design

The Stochastic Encoder/Decoder/Predictor design is illustrated in Figure 13.5. Like equation 57, this figure assumes that  $X_i$  equals  $\hat{X}_i$  plus some Gaussian white noise. However, it would be straightforward to adjust this design to predict something like  $net_i$  instead of  $X_i$ .

To implement Figure 13.5, we can go through the following sequence of calculations. First, we can plug in information from time  $t-1$  into the Predictor network, and use the Predictor network to calculate  $\hat{R}(t)$ . Then we can call on the Encoder network, which inputs  $X(t)$ , along with any or all information available to the system from time  $t-1$ . (This could, in principle, include the prediction  $\hat{R}(t)$ , which does not incorporate real information from time  $t$ .) The output of the Encoder network is a vector,  $\hat{R}$ , which is a kind of best guess of the true value of  $R$ . The next step is to generate simulated values of  $R$ ,  $\hat{R}'$ , by adding random numbers to each component  $\hat{R}_i$ . Finally, the Decoder network inputs these *simulated* values,  $\hat{R}'$ , along with information from time  $t-1$ , in order to generate a prediction of  $X$ . These calculations depend on the *weights* inside the Encoder, Decoder, and Predictor networks, and *also* on our estimates of  $\sigma_i^R$  and  $\sigma_i^X$ .

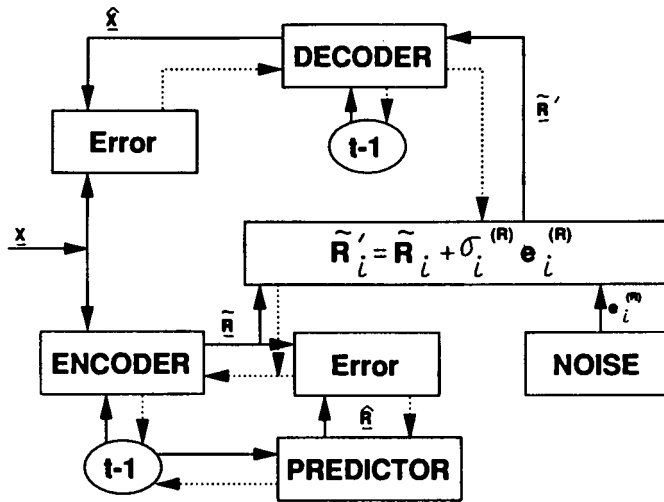


Figure 13.5 The Stochastic Encoder/Decoder/Predictor.

The Predictor network and the Decoder network in this scheme are straightforward forecasting networks that can be adapted exactly like the forecasting networks of section 5. They could even combine an upper layer of associative memory with hidden memory nodes adapted by backpropagation, if the application requires such complexity. The estimation of  $\sigma_i^x$  is straightforward. (For example,  $\sigma_i^x$  can be estimated as the observed root mean square average of  $X_i - \hat{X}_i$ .)

### 6.3. How to Adapt the System

The adaptation of the Encoder network and the parameters  $\sigma_i^R$  is more difficult. Conceptually, we try to adapt *all* parts of the network *together* so as to minimize:

$$E = \sum_i (X_i - \hat{X}_i(R'(x, \sigma^R, e^R)))^2 / (\sigma_i^x)^2 + \sum_j (\hat{R}_j - R_j(x))^2 \quad (46)$$

$$+ \sum_i \log (\sigma_i^x)^2 + \sum_j ((\sigma_j^R)^2 - \log (\sigma_j^R)^2).$$

In performing this minimization, it is critical that we account for the effect of  $\sigma^R$  and of the Encoder network in changing the errors of the Decoder network; that is why those arguments of the functions are spelled out carefully in equation 46.

In particular, this arrangement requires that we use backpropagation—gradient-based learning—to adapt the Encoder and the parameters  $\sigma_i^R$ . The gradients are calculated as shown by the dashed lines in Figure 13.5. These *dashed lines represent the calculations that feed back the prediction errors*,



used to adapt *all* parts of the network simultaneously. For the Encoder, we calculate the relevant derivative of  $E$  with respect to  $\hat{R}_j$  as:

$$F_{\hat{R}_j} = 2(\hat{R}_j - \hat{R}_j) + F_{\hat{R}_j'} \quad (47)$$

where the first term results from differentiating the  $R$ -prediction error in equation 46 with respect to  $\hat{R}_j$ , and the second term represents the derivative of the  $X$ -prediction error, calculated by backpropagation through the Decoder network back to  $\hat{R}_j'$ . Note that equation 46 requires us to divide the  $X_i$  prediction error by  $(\sigma_i^R)^2$ , before we begin the backpropagation. (To make this scheme more robust, we might want to put an arbitrary floor on what we are willing to divide by here.) To adapt the Encoder network, we then propagate the  $F_{\hat{R}}$  derivatives back through the Encoder network to the weights in that network, and adjust them accordingly.

Intuitively, equations 46 and 47 say that we adapt the Encoder network so that the resulting  $\hat{R}$  is both *predictable* from the past and *useful* in reconstructing the observed variables  $X_i$ . If we delete the Predictor network, and use this scheme as a kind of feature extractor, then these equations are really telling us to minimize the variance of  $\hat{R}$ , so as to prevent a kind of indirect bias or divergence that would otherwise sneak in.

In a similar vein, we adapt the parameters  $(\sigma_j^R)^2$  based on:

$$\frac{\partial E}{\partial (\sigma_j^R)^2} = F_{\hat{R}_j'} * e_j + \frac{\partial}{\partial (\sigma_j^R)^2} ((\sigma_j^R)^2 - \log (\sigma_j^R)^2),$$

where the right-hand term tells us to make  $\sigma_j^R$  larger, but the left-hand term will stop us from doing so once the random numbers start to cause large errors in decoding.

Whenever any of these networks makes use of information calculated within the system at time  $t - 1$ , we must propagate derivatives of equation 46 backwards to the networks which produced that information, and add them to the derivatives used to adapt these networks. All the considerations in section 5 still apply, when we try to minimize this measure of error over time.

## 6.4. Using Stochastic Models in Adaptive Critics

Section 2 and Chapter 3 include several procedures for using a Model network that require that derivatives be propagated *through* the Model network. We had a choice of two procedures, one based on a *simulation* of  $\hat{R}(t + 1)$  and the other based on using the *actual* observations from time  $t + 1$ .

Using the design in Figure 13.5, it is easy to generate a simulation of  $\hat{R}(t + 1)$ , starting from data at time  $t$ . First one uses the Predictor network to generate  $\hat{R}_j(t + 1)$ . To simulate  $\hat{R}_j(t + 1)$ , one adds  $\hat{R}_j(t + 1)$  to a random number, whose variance is set to the observed average error in predicting  $\hat{R}_j$  from  $\hat{R}_j$ . Next one generates a new set of random numbers,  $e^R$ , and generates  $\hat{R}'(t + 1)$  as shown in Figure 13.5. Finally, one uses the Decoder network to generate  $\hat{X}$  and a third set of random numbers to simulate  $X$ . In backpropagating through this structure, one simply proceeds as if the random numbers had been external inputs to the system.

To use actual observations,  $X(t + 1)$ , one plugs these observations into the Encoder network, which then outputs the actual values of  $\hat{R}(t + 1)$ . One uses these actual values in simulating  $\hat{R}'(t + 1)$ , which is then plugged into the Decoder network to generate  $\hat{X}(t + 1)$ . We still use random numbers in

simulating  $R'$ , but we use actual errors, in effect, to infer what the random numbers “actually were” in generating  $R$  and  $X$ . In backpropagating through this structure, we proceed exactly as we did before. The continuing presence of randomness in this structure reflects our uncertainty, even after the fact, about the true value of  $R(t+1)$ . This general approach follows the structure of our uncertainty, but its net effect on the consistency of adaptive critic systems has yet to be verified even through linear examples.

## 6.5. Checks of Consistency in a Linear Example: Overview

The remainder of this section will verify that the procedure given above can give the right answer, in a simple linear example that does not involve any prediction. Even in the linear example, these checks will not be totally complete. I will prove, for example, that we get the right Decoder network *if* the Encoder network and  $\sigma^x$  have already been adapted correctly. Instead of considering real-time learning effects, I will consider the equilibrium that results after an infinite number of observations are available; in other words, I will assume that our estimates are all based on the *true* expectation values of all observable quantities. I will not consider issues of uniqueness and stability for the whole system, even in the linear case. In summary, these are really just consistency checks; they leave open a lot to be done in future research. In actuality, it has taken considerable effort to find a system that passes these basic tests, and I have been surprised to see how many realistic-looking designs do not.

For all these tests, I will assume that we are observing a system governed by:

$$x(t) = AR(t) + e(t), \quad (48)$$

where:

$$\langle RR^T \rangle = I \quad (49)$$

and:

$$\langle e_i(t)e_j(t) \rangle \triangleq S_{ij}^x = I_{ij}(\sigma_i^x)^2, \quad (50)$$

where  $R_i(t)$  and  $e_i(t)$  are independent normal random variables. This system is *not* a true dynamic system, nor is it anything like a realistic plant; however, it does capture the essence of the difficulty here. (Adding dynamics basically just changes the expected values or means from which the probability distributions are calculated.) There is no loss of generality in assuming unit variance for the unobserved variables  $R$ , as in equation 49, since we can always rescale them without changing any observable characteristics of the model. Equation 50 reflects our initial approach in equation 44; the term on the right is not an identity matrix, but a mathematician's way of representing a diagonal matrix.

To begin with, I will assume that our network is adapted to minimize  $E$ , as defined in equation 46, *except* that I replace:

$$\sum_j ((\sigma_j^R)^2 - \log(\sigma_j^R)^2)$$

by:

$$Tr(S^R) - \log \det S^R,$$

where  $S^R$  is the covariance *matrix* used to generate the random vector  $e^R$ . (Note that my notation for  $e^R$  here is slightly different from what it is in Figure 13.5.) I will assume that the “neural network” takes the form:

$$R = \beta x \quad (51)$$

$$R' = \beta x + e^R \quad (52)$$

$$\hat{x} = \alpha R' = \alpha(\beta x + e^R) \quad (53)$$

$$\langle e^R (e^R)^T \rangle = S^R. \quad (54)$$

In the actual operation of this system, the minimization of  $E$  would be used to adapt the Encoder and Decoder *simultaneously*; however, for simplicity, I will only carry out certain easier tests that are necessary but not sufficient (though they are highly suggestive) in showing that the system will work properly. More precisely, I will demonstrate, in order, that: (1) for a certain value of  $S^R$  and a correct Encoder, we adapt the weights in the Decoder correctly; (2) for a correct Decoder, we adapt the weights of the Encoder correctly; (3) with a correct Decoder, we adapt the values of  $S^R$  to the values required in part (1). In all cases, I will assume that the parameters  $\sigma_i^T$  have already been estimated correctly. In these calculations, it will turn out that a correct form of  $S^R$  is a matrix that need not be diagonal. However, one can always *rotate* the vector  $R$  so as to make that matrix diagonal; for that reason, efforts to minimize  $E$  as defined in equation 46 will always give us a result that is correct within the more general matrix framework. This fine point will be explained in more detail below.

There is no way to do these calculations without heavy use of matrix calculus and statistics.

Before checking for the correctness of what we adapt, we must first decide what the correct values *are*. Comparing equation 47 to equation 53, we clearly want:

$$\alpha = A. \quad (55)$$

A correct Encoder should output the best estimate of  $R(t)$  given knowledge of  $x(t)$ , based on knowledge of equation 48. We can calculate this best estimate,  $\beta x(t)$ , simply by regressing  $R$  on  $x$ , using classical methods [15] that are known to minimize square error:

$$\beta = \langle R x^T \rangle (\langle x x^T \rangle)^{-1}.$$

To work this out, note that our original linear model implies:

$$\begin{aligned} \langle x x^T \rangle &= \langle (AR + e)(AR + e)^T \rangle \\ &= \langle A R R^T A^T \rangle + \langle e e^T \rangle = A A^T + S^R. \end{aligned} \quad (56)$$

(Note that our assumption that  $e$  is random implies  $\langle ex \rangle = \langle eR \rangle = 0$ .) Likewise:

$$\langle Rx^T \rangle = \langle R(AR + e)^T \rangle = \langle RR^T A^T \rangle = A^T.$$

Substituting this into our formula for  $\beta$ , we deduce that the correct value is:

$$\beta = A^T(AA^T + S^x)^{-1}. \quad (57)$$

## 6.6. Checking the Adaptation of the Decoder

For this check, we assume that the Encoder is already adapted corrected, as defined by equation 57:

$$\beta = A^T(AA^T + S^x)^{-1}. \quad (58)$$

We also assume that  $S^R$  has been adapted to the following value:

$$S^R = I - A^T(AA^T + S^x)^{-1}A. \quad (59)$$

Our task here is to figure out what the Decoder will be adapted to.

Looking at equation 46, we can see that the Decoder is simply being adapted to minimize square error in predicting  $x(t)$  from  $R'(t)$ . From classical statistics [15], we know what this results in, in the linear case:

$$\alpha = \langle xR'^T \rangle (\langle R'R'^T \rangle)^{-1}.$$

Clearly this will lead to a unique solution for  $\alpha$ . We want to verify that it will lead to the correct solution; in other words, we want to verify:

$$\begin{aligned} A &= \langle xR'^T \rangle (\langle R'R'^T \rangle)^{-1} \\ &= \langle x(\beta x + e^R)^T \rangle (\langle (\beta x + e^R)(\beta x + e^R)^T \rangle)^{-1} \\ &= \langle xx^T \rangle \beta^T (\beta \langle xx^T \rangle \beta^T + S^R)^{-1}. \end{aligned} \quad (60)$$

This is equivalent to:

$$A(\beta \langle xx^T \rangle \beta^T + S^R) = \langle xx^T \rangle \beta^T.$$

Substituting in from equations 56 and 58, and recalling that  $AA^T$  and  $S^x$  are symmetrical, we can see that this is equivalent to:

$$AS^R + A(A^T(AA^T + S^x)^{-1}(AA^T + S^x)(AA^T + S^x)^{-1}A) = (AA^T + S^x)((AA^T + S^x)^{-1}A),$$

which reduces to:

$$AS^R + AA^T(AA^T + S^x)^{-1}A = A. \quad (61)$$

If we multiply equation 59 on the left by  $A$ , we can easily see that equation 61 will be true. Since equation 61 is equivalent to equation 60, which we were trying to verify, we have succeeded in verifying what we wanted to—that the Decoder weights will, in fact, be adapted to equal the correct values,  $A$ .

## 6.7. Checking the Adaptation of the Encoder

For this test, we assume that the Decoder has been adapted to the correct value, such that  $\alpha = A$ . We also assume that the  $S^x$  matrix of equation 50 is correct, and that there is no Prediction network (i.e., that  $\hat{R} = 0$ ). In this case, we are trying to minimize those terms in equation 46 that are affected by the encoder matrix  $\beta$ . We are minimizing the following effective error function with respect to  $\beta$ :

$$\begin{aligned} E_{eff} &= \langle (x - \hat{x})^T S^{x-1} (x - \hat{x}) \rangle + \langle \tilde{R}^T \tilde{R} \rangle \\ &= \langle (x - A\beta x - e^R)^T S^{x-1} (x - A\beta x - e^R) \rangle + \langle \tilde{R}^T \tilde{R} \rangle \\ &= \langle (x - A(\beta x + e^R))^T (S^x)^{-1} (x - A(\beta x + e^R)) \rangle + \langle (\beta x)^T (\beta x) \rangle \\ &= \langle x^T (I - A\beta)^T (S^x)^{-1} (I - A\beta) x \rangle + \langle x^T \beta^T \beta x \rangle + \text{terms not affected by } \beta. \end{aligned}$$

Thus, we want to minimize the following matrix trace with respect to  $\beta$ :

$$Tr(\langle xx^T \rangle + ((I - A\beta)^T (S^x)^{-1} (I - A\beta) + \beta^T \beta)). \quad (62)$$

We can simplify this minimization problem slightly by defining matrices:

$$\begin{aligned} Z &= \langle xx^T \rangle \\ C &= (S^x)^{-1} A \\ Q &= A^T (S^x)^{-1} A + I. \end{aligned}$$

Using this notation, we are trying to minimize the terms in equation 62 that depend on  $\beta$ :

$$E_{eff} = -2 \sum_{i,j,k} Z_{ij} C_{jk} \beta_{ki} + \sum_{i,j,k,l} Z_{ij} \beta_{jk}^T Q_{kl} \beta_{li}. \quad (63)$$

Differentiating equation 63 with respect to every matrix element  $\beta_{ij}$ , and collecting terms carefully, we arrive at the following condition for a minimum:

$$0 = -2C^T Z + Q\beta Z + Q\beta Z$$

$$\therefore C^T = Q\beta.$$

Going back to our definitions, this yields a solution for  $\beta$  of:

$$\beta = Q^{-1}C^T = (A^T(S^x)^{-1}A + I)^{-1}A^T(S^x)^{-1}. \quad (64)$$

In summary, our Encoder will adapt to the values shown in equation 64. Our problem is to verify that these values are the same as the correct values, the values given in equation 57. In other words, we must verify that:

$$A^T(AA^T + S^x)^{-1} = (A^T(S^x)^{-1}A + I)^{-1}A^T(S^x)^{-1}. \quad (65)$$

To verify this, we may simply multiply both sides on the right by  $(AA^T + S^x)$  and on the left by  $A^T(S^x)^{-1}A + I$ . This shows that equation 65 is equivalent to:

$$(A^T(S^x)^{-1}A + I)A^T = A^T(S^x)^{-1}(AA^T + S^x), \quad (66)$$

which is easily seen to be true. In short, since equation 65 is equivalent to equation 66, and equation 66 is true, we may deduce that equation 65 is also true. Thus, the weights we adapt to (given in equation 64) are, in fact, equal to the correct weights as defined by equation 57.

## 6.8. Checking the Adaptation of $S^R$

For our final check, we again assume that the Encoder is correctly adapted, so that  $\alpha = A$ . To figure out what  $S^R$  will be adapted to, we once again begin by figuring out what will be minimized as a function of  $S^R$ . As in our derivation of equation 62, we calculate the terms in  $\langle E \rangle$  affected by  $S^R$ :

$$\begin{aligned} E_{eff} &= \langle (x - \hat{x})^T (S^x)^{-1} (x - \hat{x}) \rangle + Tr(S^R) - \log \det S^R \\ &= \langle (e^R)^T A^T (S^x)^{-1} A e^R \rangle + Tr(S^R) - \log \det S^R + \text{terms not affected by } S^R. \end{aligned}$$

This reduces to an effective error function to be minimized of:

$$E_{eff} = Tr(S^R M) - \log \det S^R, \quad (67)$$

where I have defined:

$$M = A^T (S^x)^{-1} A + I. \quad (68)$$

From matrix calculus (in a form that recurs quite often in classical statistics [14,22]), we know that the minimum of equation 67 occurs when:

$$S^R = M^{-1}. \quad (69)$$

Our basic task in this section is to verify that  $S^R$  we adapt to—which is given in equations 68 and 69—will, in fact, equal the value we required earlier, in equation 59. In other words, we need to verify that:

$$(A^T(S^R)^{-1}A + I)^{-1} = I - A^T(AA^T + S^R)^{-1}A. \quad (70)$$

To verify this, we simply multiply the right-hand side of equation 70 by  $I + A^T(S^R)^{-1}A$ , and verify that we arrive at the identity matrix. When we do this, we get:

$$\begin{aligned} & (I + A^T(S^R)^{-1}A)(I - A^T(AA^T + S^R)^{-1}A) \\ &= I + A^T(S^R)^{-1}A - A^T(AA^T + S^R)^{-1}A - A^T(S^R)^{-1}AA^T(AA^T + S^R)^{-1}A \\ &= I + A^T(S^R)^{-1}A - A^T(I + (S^R)^{-1}AA^T)(AA^T + S^R)^{-1}A \\ &= I + A^T(S^R)^{-1}A - A^T(S^R)^{-1}(S^R + AA^T)(AA^T + S^R)^{-1}A \\ &= I + A^T(S^R)^{-1}A - A^T(S^R)^{-1}A = I, \end{aligned}$$

exactly as required.

To complete this discussion, we need to comment on the diagonal form of  $S^R$ , which our actual neural network design requires. We have just proven that a modified form of  $E$ —allowing for arbitrary  $S^R$ —is minimized for appropriate combinations of  $\alpha$ ,  $\beta$ , and  $S^R$ . (This modified form reduces to the original form when  $S^R$  is, in fact, diagonal.) However, we know that there are equivalent ways to represent the exact same stochastic model, simply by rotating the vector  $R$ . Such a rotation does not affect the matrix  $\langle RR^T \rangle = I$ . Since  $A^T A$  is a positive definite real matrix, we know that we can always arrive at an equivalent form of the model that diagonalizes  $A^T A$ , and therefore diagonalizes  $S^R$ . Therefore, if we try to minimize  $E$  *subject to the constraint* that  $S^R$  be diagonal, we know that one of the unconstrained optimal solutions will still be able to satisfy that constraint. Thus, in building a system that minimizes  $E$  subject to that constraint, we are certain that we still can arrive at a solution that solves the unconstrained problem as well. No constrained optimum can lead to an  $E$  lower than that of an unconstrained optimum; therefore, we can be sure that a solution equivalent (within rotation) to the correct solution will emerge as a minimum of the constrained problem.

As stated above, there is still more theory to be developed, even in the linear case, but this does appear to be a promising beginning in an area that has been largely neglected.

## 6.9. Implications for Realism versus Phenomenology, Fuzzy versus Bayesian, Etc.

The design in Figure 13.5 has serious implications for some very old arguments. There are at least three competing theories for how the mammalian brain represents uncertainty:

1. That it does not really represent uncertainty at all—that it uses methods like those of section 5 to build up any implicit representation of uncertainty
2. That it uses methods like those described here, which account for cross-correlations at the same time  $t$ , but use methods like those of section 5 (in adapting the Predictor and Encoder) to represent more abstract patterns of correlation over time
3. The radical Bayesian or realistic theory, in which *all* “short-term” memory takes the form of  $R$  vectors with uncertainty factors attached

These choices have their parallel in engineering. The first approach is incapable of representing the statistics of a general vector ARMA process. The second approach is close to the approach used most by statisticians [15,22] in describing vector ARMA processes; in that formulation, these processes are completely and uniquely identifiable. The third approach is commonly used by engineers [23] in describing such processes.

In Kalman filtering [23], the state of the plant is summarized in one vector,  $\hat{R}$  (usually denoted as “ $x$ ”). This state estimate (and its covariance matrix) incorporates *all* memory of the plant. At each time  $t$ , the current state estimate is based on the forecasts,  $\hat{R}$ , and the current observations  $X(t)$ . There are forecasts corresponding to *all* state variables, and *all* state variables are updated by methods similar to those of this section.

The methods of this section could be used to implement a radically realistic design, similar to Kalman filtering. To do this, one simply forbids the three networks from using any internal memory units. The Encoder and Decoder networks would be forbidden from using any information from time  $t - 1$  except for  $\hat{R}$ . As a practical matter, it would seem more sensible to *limit* such memory units rather than *forbid* them altogether; the challenge would be to develop criteria for keeping them or deleting them (or adding them to  $R$ ) *after* the fact, based on an empirical measure of their performance. Research on these topics has hardly begun.

The biological evidence on these choices is also unclear as yet, because there are several types of pyramid cells in the cerebral cortex and other cells in the thalamus that could plausibly be involved in the different schemes.

The issue described here concerns realism versus phenomenology as principles used by the *brain*. This is logically quite distinct from the parallel debate concerning objective reality in the *physical universe*, a debate that also remains unresolved [24].

## 13.7. CONCLUSIONS

Research approaches and preliminary designs exist for bridging most of the gap between neurocontrol as it is used today and the form of neurocontrol that will be needed in order to understand (and replicate) true intelligent control as it exists in the brain. To fulfill this potential will take considerably more work on the part of many researchers working in many different areas, and drawing on many different disciplines, but the scientific importance of the work is enormous. With proper motivation and encouragement, control engineers, cooperating with biologists, could play a decisive role in making it possible to understand human learning and the human mind.



### 13.8. REFERENCES

- [1] P. Werbos, The cytoskeleton: Why it may be crucial to human learning and to neurocontrol. *Nanobiology*, 1:(1), 1992.
- [2] P. Werbos, Neurocontrol, biology and the mind. In *IEEE/SMC Proceedings*, IEEE, New York, 1991.
- [3] P. Werbos, A menu of designs for reinforcement learning over time. *Neural Networks for Control*, W. T. Miller, R. Sutton, and P. Werbos, eds., MIT Press, Cambridge, MA, 1990.
- [4] P. Werbos, Building and understanding adaptive systems: a statistical/numerical approach to factory automation and brain research. *IEEE Trans. Systems, Man, and Cybernetics*, 17:(1), January–February 1987.
- [5] Commons, Grossberg, and Staddon, eds., *Neural Network Models of Conditioning and Action*, Erlbaum, Hillsdale, NJ, 1991.
- [6] A. Barto, R. Sutton, and C. Anderson, Neuronlike elements that can solve difficult learning control problems. *IEEE Trans. Systems, Man, and Cybernetics*, 13:(5), 1983.
- [7] P. Werbos, Advanced forecasting methods for global crisis warning and models of intelligence. *General Systems Yearbook*, 1977.
- [8] P. Werbos, Consistency of HDP applied to a simple reinforcement learning problem. *Neural Networks*, 3:179–189, October 1990.
- [9] G. Lukes, B. Thompson, and P. Werbos, Expectation driven learning with an associative memory. In *Proceedings of the International Joint Conference on Neural Networks* (Washington, D.C.), Erlbaum, Hillsdale, NJ, January 1990.
- [10] M. Jordan and R. Jacobs, Learning to control an unstable system with forward modeling. *Advances in Neural Information Processing Systems 2*, D. Touretzky, ed., Morgan Kaufmann, San Mateo, CA, 1990.
- [11] D. Sofge and D. White, Neural network based process optimization and control. In *IEEE Conference on Decision and Control* (Hawaii), IEEE, New York, 1990.
- [12] K. Narendra and Annaswamy, *Stable Adaptive Systems*, Prentice Hall, Englewood Cliffs, NJ, 1989.
- [13] R. Howard, *Dynamic Programming and Markov Processes*, MIT Press, Cambridge, MA, 1960.

- [14] P. S. Goldman-Rakic, The Nervous System. *Handbook of Physiology*, F. Plum, ed., section 1, Vol. 5, *Higher Functions of the Brain*, Part 1, p. 373, Oxford University Press, New York, 1987.
- [15] T. H. Wonnacott and R. Wonnacott, *Introductory Statistics for Business and Economics*, 2nd ed. John Wiley & Sons, New York, 1977.
- [16] P. Werbos, Neurocontrol and fuzzy logic: Connections and designs. *International Journal on Approximate Reasoning*, February 1992. See also P. Werbos, Elastic fuzzy logic: a better fit to neurocontrol. In *I-zuka-92 Proceedings*, Japan, 1992.
- [17] M. L. Minsky and S. A. Papert, *Perceptrons*, MIT Press, Cambridge, MA, 1969.
- [18] C. Parten, R. Pap, and C. Thoma, Neurocontrol applied to telerobotics for the space shuttle. *Proceedings of the International Neural Network Conference* (Paris, 1990), Vol. I. Kluwer Academic Press.
- [19] P. Werbos, Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1:339–356, October 1988.
- [20] J. Schmidhuber, Recurrent networks adjusted by adaptive critics. In *IJCNN Proceedings*, p. I-719, op. cit. [9].
- [21] K. Joreskog and Sorbom, *Advances in Factor Analysis and Structural Equation Models*. University Press of America, Lanham, MD. (In actuality, this chapter owes more to an out-of-print book by Maxwell and Lawley on factor analysis as a maximum likelihood method.)
- [22] E. J. Hannan, *Multiple Time-Series*, John Wiley & Sons, New York, 1970.
- [23] A. Bryson and Y. Ho, *Applied Optimal Control: Optimization, Estimation and Control*, Hemisphere, 1975.
- [24] P. Werbos, Bell's theorem: the forgotten loophole. *Bell's Theorem, Quantum Theory and Conceptions of the Universe*, M. Kafatos, ed., Kluwer Academic Press, 1989.
- [25] P. Werbos and A. Pellionisz, Neurocontrol and neurobiology: New developments and connections. In *Proceedings of the International Joint Conference on Neural Networks*, IEEE, New York, 1992.