HANDBOOK OF INTELLIGENT CONTROL

NEURAL, FUZZY, AND ADAPTIVE APPROACHES

Edited by
David A. White
Donald A. Sofge

1992


VAN NOSTRAND REINHOLD
New York

FOREWORD [1]

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.

---

1 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.

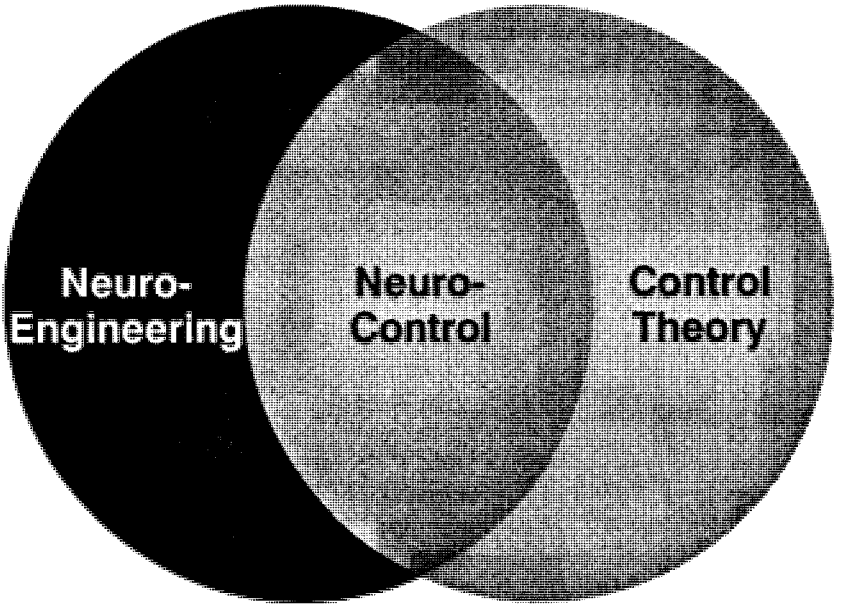

Figure F.1 Neurocontrol as a subset.

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

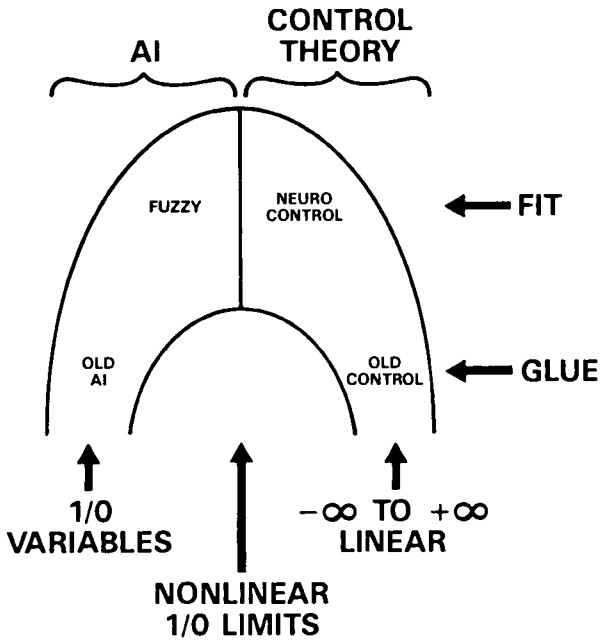

Figure F.2 Aspects of intelligent control.

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 ±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.*

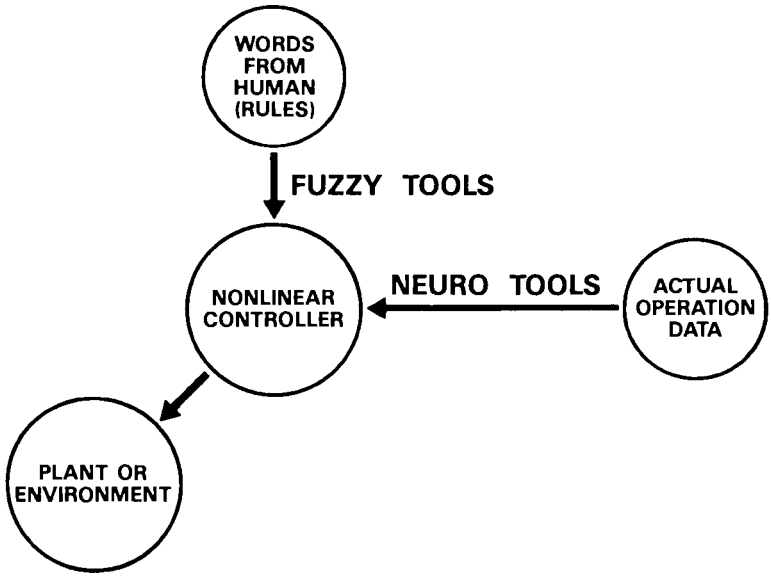

Figure F.3 A way to combine fuzzy and neural tools.

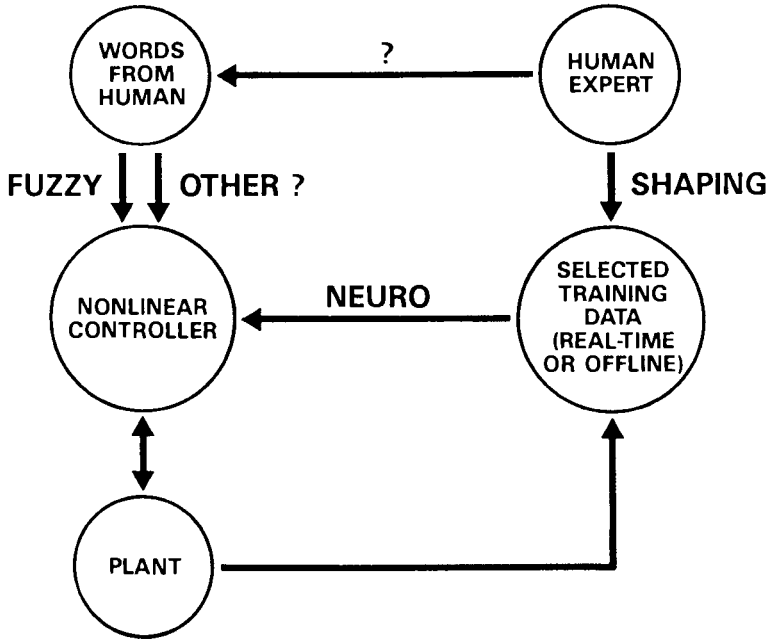

Figure F.4 A way to combine fuzzy and neural tools.

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

3

NEUROCONTROL AND SUPERVISED LEARNING: AN OVERVIEW AND EVALUATION

Paul J. Werbos
NSF Program Director for Neuroengineering
Co-director for Emerging Technologies Initiation[1]

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

1 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, $\boldsymbol{X}(t)$, to a vector of targets or desired outputs, $\boldsymbol{Y}(t)$. "Learning" means adjusting a vector of weights or parameters, $\boldsymbol{W}$, in the ANN. "Batch learning" refers to any method for adapting the weights by analyzing a fixed database of $\boldsymbol{X}(t)$ and $\boldsymbol{Y}(t)$ 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 $\boldsymbol{X}(t)$ and then $\boldsymbol{Y}(t)$ 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 $\boldsymbol{X}$ to $\boldsymbol{Y}$. In theory, we could try to do supervised learning for *any* function or mapping f such that:

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

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 $\boldsymbol{X}$ to $\boldsymbol{Y}$ is modeled by the following equations:

$$x_i = X_i \qquad 1 \le i \le m \tag{2}$$

$$net_i = \sum_{j=1}^{i-1} W_{ij}\, x_j \qquad m < i \le N + n \tag{3}$$

$$x_i = 1/(1 + \exp(-net_i)) \qquad m < i \le N + n \tag{4}$$

$$\hat{Y}_i = x_{i+N} \qquad 1 \le i \le n, \tag{5}$$

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 ($\boldsymbol{X}$) are usually sensor readings and the targets ($\boldsymbol{u}^*$) are *desired* control responses to those sensor readings. (In this application, the targets $\boldsymbol{Y}$ happen to be control signals $\boldsymbol{u}^*$.)

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

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

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

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:

$$W_{ij}(T+1) = W_{ij}(t) - LR * F\_W_{ij}(t), \tag{7}$$

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.

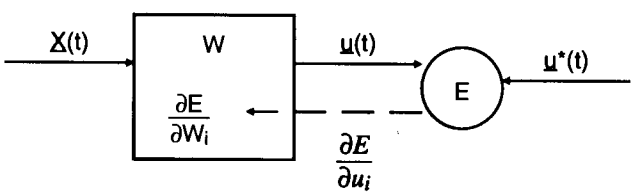

Figure 3.1 Basic backpropagation in supervised control.

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:

$$W_{ij}(t+1) = W_{ij}(t) - LR(t) * S_{ij} * F\_W_{ij}(t), \tag{8}$$

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]:

$$LR(t+1) = a*LR(t) + b*LR(t)*\left(\frac{F\_W(t+1) \cdot F\_W(t)}{|F\_W(t)|^2}\right), \tag{9}$$

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:

$$F\_\hat{Y}_i(t) = \frac{\partial E(t)}{\partial \hat{Y}_i(t)} = \hat{Y}_i(t) - Y_i(t) \tag{10}$$

$$\{F\_W_{ij}(t)\} = F\_f_W(X, W, F\_\hat{Y}), \tag{11}$$

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:

$$F\_x_i(t) = F\_\hat{Y}_{i-N}(t) + \sum_{j=i+1}^{N+m} W_{ji} * F\_net_j(t) \qquad i = N+m, \ldots, 1 \tag{12}$$

$$F\_net_i(t) = s'(net_i(t)) * F\_x_i(t) \qquad i = N+m, \ldots, m+1 \tag{13}$$

$$F\_W_{ij}(t) = F\_net_i(t) * x_j(t), \qquad all\ W_{ij} \tag{14}$$

where:

$$s'(net_i(t)) = x_i(t) * (1 - x_i(t)), \tag{15}$$

where the first term on the right-hand side of equation 12 is taken as zero for $i \le N$, and where F_net_j is taken as zero for $j \le m$. 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 $i \le m$); 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:

$$F\_z = F\_f_z(z, F\_\hat{Y}) \tag{16}$$

$$\{F\_W_{ij}\} = F\_g_W(X, W, F\_z).$$

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:

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

which is computed by iterating over the equation:

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

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 = ∞. 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 $F\_\hat{Y}$ 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:

$$F\_y^{(n+1)} = F\_\hat{Y} + F\_f_y(\hat{Y}, X, W, F\_y^{(n)}). \tag{19}$$

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 nth *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:

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

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

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:

$$f(X, y, W_x, W_y) = W_X X + W_y y . \tag{22}$$

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):

$$y^{(\infty)} = W_X X + W_y y^{(\infty)} \tag{23}$$

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

$$F\_f_y(\hat{Y}, X, W_X, W_y, F\_y^{(\infty)}) = F\_\hat{Y} + W_y^T F\_y^{(\infty)} = (I - W_y^T)^{-1} F\_\hat{Y} \tag{25}$$

$$F\_f_X(\hat{Y}, X, W_X, W_y, F\_\hat{Y}) = W_X^T F\_y^{(\infty)} = W_X^T (I - W_y^T)^{-1} F\_\hat{Y} \tag{26}$$

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*—$\boldsymbol{X}(t)$—specified by a higher-level controller or a human, and output the control signals, $\boldsymbol{u}(t)$, 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 θ_1 and θ_2. If we ignore dynamic effects, then it is reasonable to assume that x_1 and x_2 are a function, f, of θ_1 and θ_2, as shown in the figure. Furthermore, if f happens to be invertible (i.e., if we can solve for θ_1 and θ_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 θ_1 and θ_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* $\boldsymbol{X}(t)$ and $\boldsymbol{u}(t)$ simply by flailing the arm about; they train a neural net to learn the inverse mapping from $\boldsymbol{X}(t)$ to $\boldsymbol{u}(t)$. In other words, they use supervised learning, with the actual $\boldsymbol{X}(t)$ used as the inputs and the actual $\boldsymbol{u}(t)$ 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].)

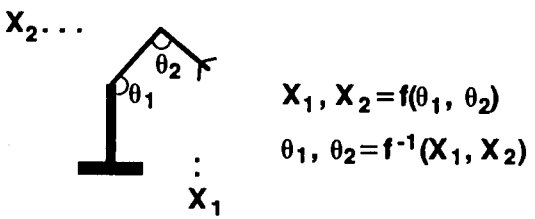

Figure 3.2 Direct inverse control.

- 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 $\theta(t)$ 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):

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

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:

$$x(t+1) = f(x(t), u(t)). \tag{28}$$

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 $\mathbf{F\_x}(T+1) = F\_U_x(x(T+1))$ (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. $\mathbf{F\_u}(t) = F\_f_u(x(t), u(t), F\_x(t+1))$ (i.e., backpropagate through the model).

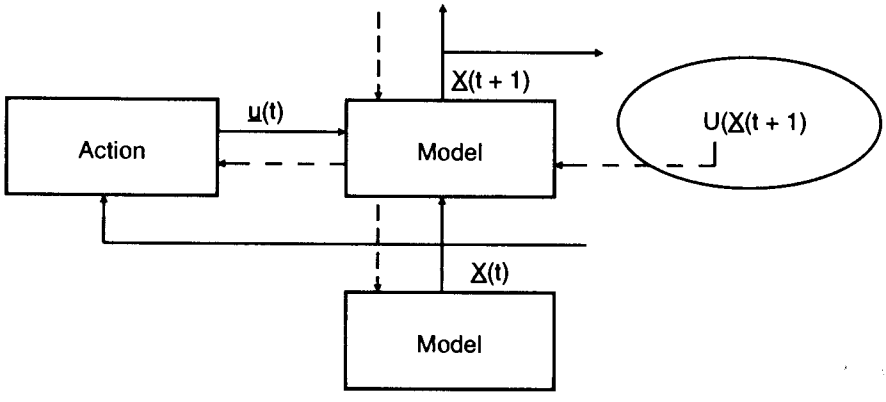

Figure 3.3 Backpropagating utility, using backpropagation through time (BTT).

b. $\mathbf{F\_x}(t) = F\_U_x(\boldsymbol{x}(t)) + F\_f_x(\boldsymbol{x}(t), \boldsymbol{u}(t), F\_x(t+1))$ (backpropagate through the model, adding grad U).
c. $\mathbf{F\_W} = \mathbf{F\_W} + F\_A_W(\boldsymbol{x}(t), \mathbf{F\_u}(t))$ (propagate derivatives through the Action network).

6. Update **W** = **W** + learning_rate * **F_W**.

Step 5 is a straightforward application of the chain rule for ordered derivatives, discussed in Chapter 10. All of the "F_" 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—$F\_A_x(\boldsymbol{x}(t), F\_u(t))$—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 $\boldsymbol{X}$ and the outputs $\boldsymbol{R}$ 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.

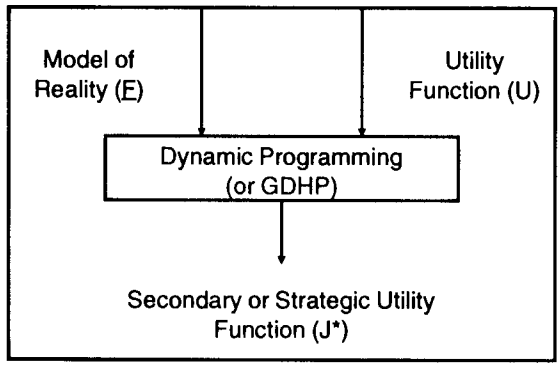

Figure 3.4 Inputs and outputs of dynamic programming.

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, $\boldsymbol{F}$, 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 $\boldsymbol{R}(t)$, where $\boldsymbol{R}$ 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 $J(\boldsymbol{R}(t))$ [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 $J(\boldsymbol{R}(t))$ [16,35]. The derivative of $J(\boldsymbol{R}(t))$ with respect to $R_i(t)$ is usually written as $\lambda_i(\boldsymbol{R}(t))$.
3. Globalized DHP (GDHP), which adapts a Critic network whose output is an approximation of $J(\boldsymbol{R}(t))$, 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 $\boldsymbol{R}(t)$, but not the vector of actions, $\boldsymbol{u}(t)$, 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:

$$J(\boldsymbol{R}(t)) = \max_{\boldsymbol{u}(t)} \left(U(\boldsymbol{R}(t),\boldsymbol{u}(t)) + \langle J(\boldsymbol{R}(t+1))\rangle/(1+r) - U_0\right), \tag{29}$$

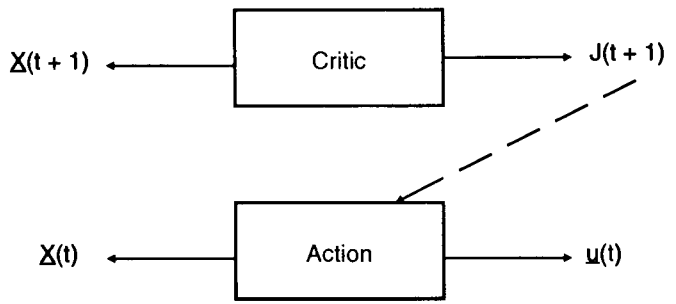

Figure 3.5 The two-network design of Barto, Sutton, and Anderson.

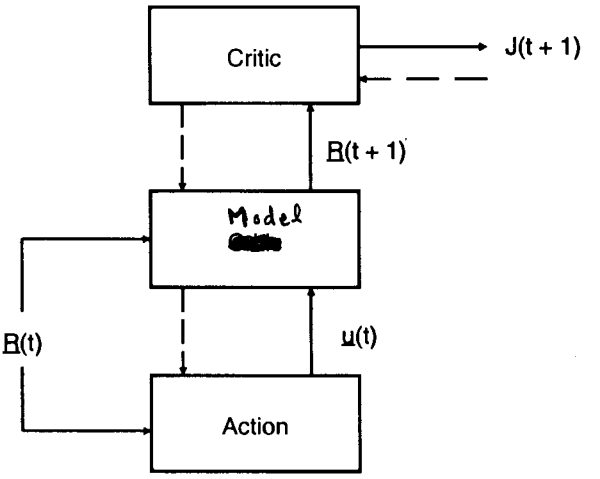

Figure 3.6 The Backpropagated Adaptive Critic (with J-style Critic).

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:

$$J'(\boldsymbol{R}(t),\boldsymbol{u}(t)) = U(\boldsymbol{R}(t),\boldsymbol{u}(t)) + \langle J(\boldsymbol{R}(t+1))\rangle/(1+r) \tag{30}$$

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

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

Both Watkins and our group developed a way to adapt a Critic network that inputs $\boldsymbol{R}$(t) and $\boldsymbol{u}$(t), and outputs an estimate of $J'(\boldsymbol{R}(t),\boldsymbol{u}(t))$. 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

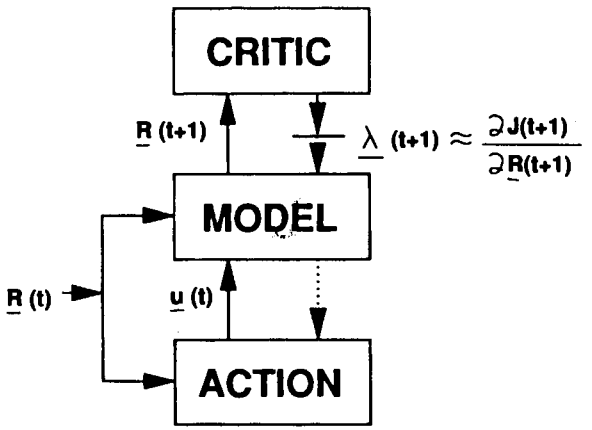

Figure 3.7 The Backpropagated Adaptive Critic (with λ-style Critic and U independent of $u(t)$).

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 $J(\boldsymbol{R}(t))$. 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 $\boldsymbol{u}(t)$ 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 $\boldsymbol{u}$ 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 $\boldsymbol{u}(t)$ 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 $\boldsymbol{R}(t+1)$ is input to the Critic at time t + 1) or in a "dreaming" or exploratory mode (where $\boldsymbol{R}(t)$ is chosen "at random" and $\boldsymbol{R}(t+1)$ 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:

$$\begin{aligned} F\_\boldsymbol{R}(t+1) &= F\_J_{\boldsymbol{R}}(\boldsymbol{R}(t+1)) \\ F\_\boldsymbol{u}(t) &= F\_f_{\boldsymbol{R}}(\boldsymbol{R}(t), \boldsymbol{u}(t), F\_R(t+1)) \\ F\_W &= F\_A_W(\boldsymbol{R}(t), W, F\_\boldsymbol{u}(t)). \end{aligned} \tag{32}$$

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

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 $\boldsymbol{u}$ *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 $J'(\boldsymbol{R}(t),\boldsymbol{u}(t))$ *directly* as a function of $\boldsymbol{u}(t)$, 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 $\boldsymbol{X}(t+1)$ may depend on phenomena we observed a few time cycles ago that happen to be out of current sensor range.

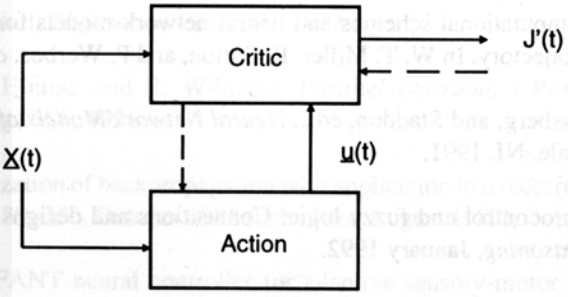

Figure 3.8 Adapting the Action network with ADAC.

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 Markhov 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.