HANDBOOK OF INTELLIGENT CONTROL

NEURAL, FUZZY, AND ADAPTIVE APPROACHES

Edited by
David A. White
Donald A. Sofge

1992

VNR 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.

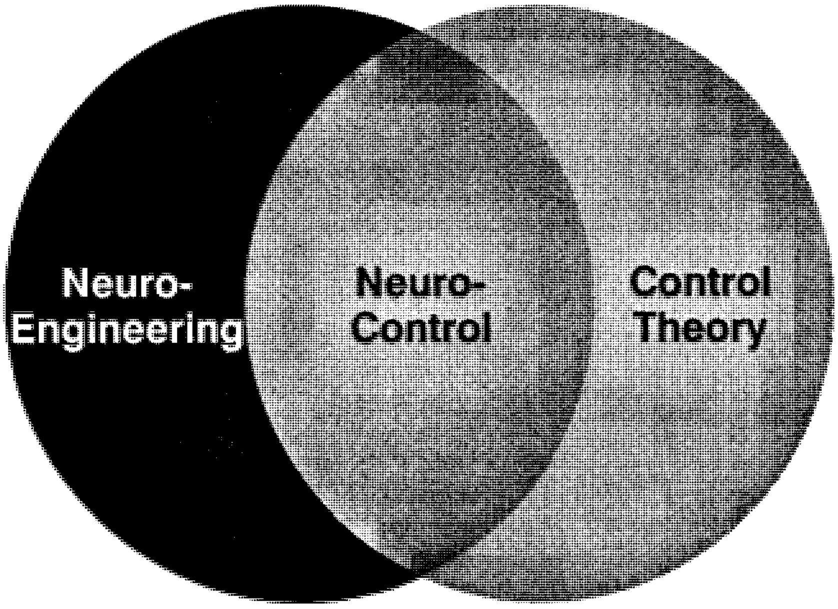

Figure F.1 Neurocontrol as a subset.

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

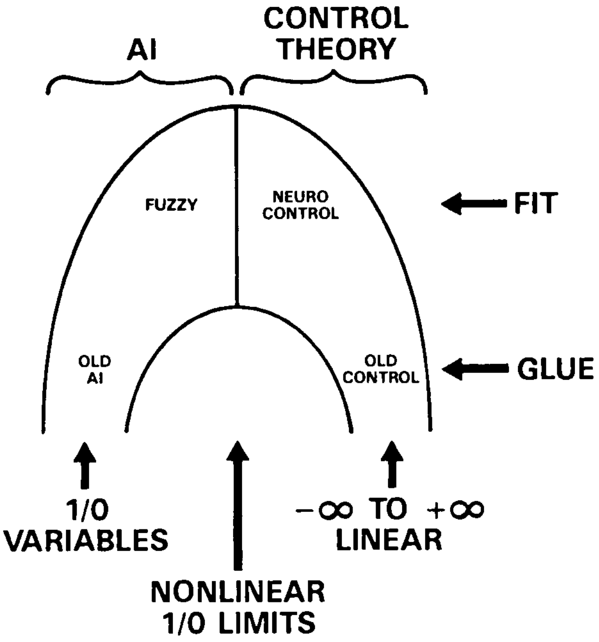

Figure F.2 Aspects of intelligent control.

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.*

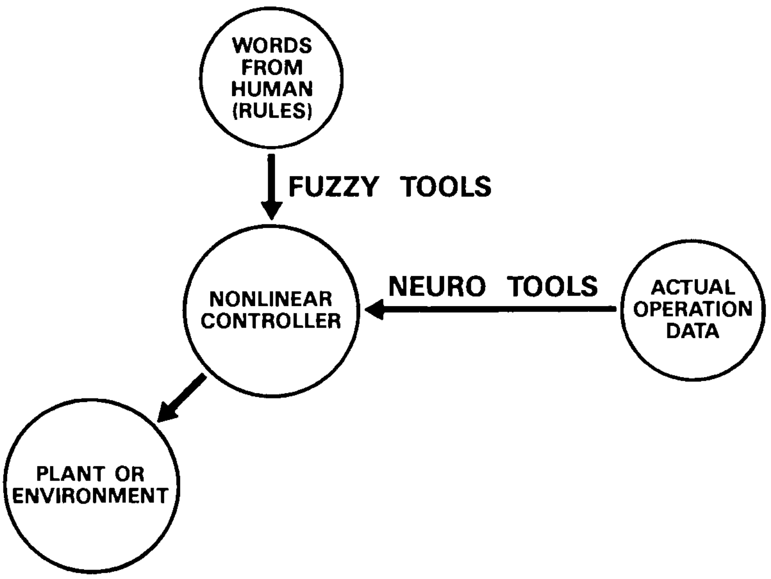

Figure F.3 A way to combine fuzzy and neural tools.

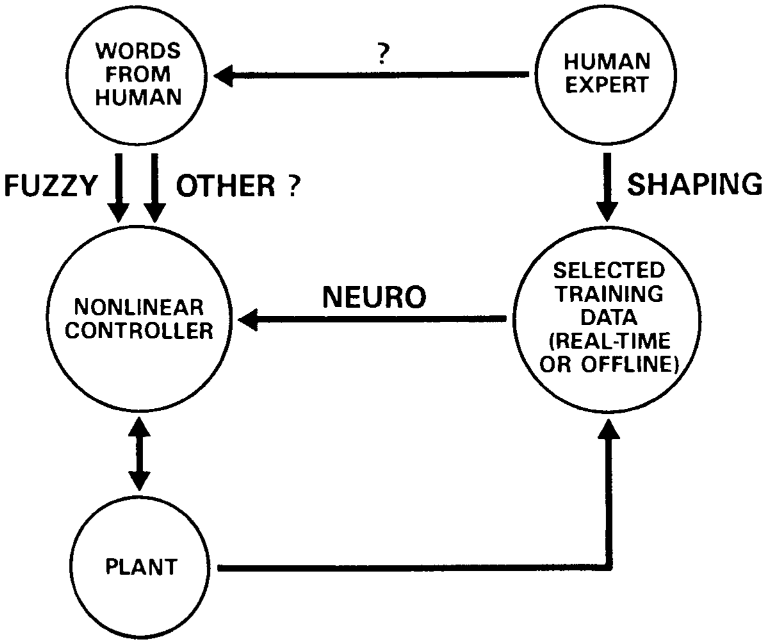

Figure F.4 A way to combine fuzzy and neural tools.

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

13

APPROXIMATE DYNAMIC PROGRAMMING FOR REAL-TIME CONTROL AND NEURAL MODELING

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

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:

1 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***(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:

$$F\_f_x(arguments,\ F\_Y) \triangleq \text{the vector} \sum_i F\_Y_i * \frac{\partial f_i}{\partial x_j}(arguments)\,. \tag{1}$$

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*, 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, $\mathbf{R}(t)$, becomes available.
2. The controller performs its various calculations and invokes the Action network A to calculate the actions $\mathbf{y}(t) = A(\mathbf{R}(t))$. Total utility, $U(\mathbf{R}(t),\mathbf{u}(t))$ is also calculated.
3. The action $\mathbf{u}(t)$ 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:

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

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 $\boldsymbol{u}(t)$. One way is simply to define the *function* $\boldsymbol{u}(\boldsymbol{R}(t))$ as that function of $\boldsymbol{R}$ which, for every $\boldsymbol{R}$, maximizes the right-hand side of the Bellman equation. With that definition (for the case $r = 0$), the Bellman equation becomes:

$$J(\boldsymbol{R}(t)) = U(\boldsymbol{R}(t), \boldsymbol{u}(\boldsymbol{R}(t)) + < J(\boldsymbol{R}(t+1)) > - U_0 . \tag{3}$$

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

$$\begin{aligned}\lambda_i(\boldsymbol{R}(t)) &\overset{\Delta}{=} \frac{\partial J(\boldsymbol{R}(t))}{\partial R_i(t)} = \frac{\partial}{\partial R_i(t)} U(\boldsymbol{R}(t), \boldsymbol{u}(\boldsymbol{R}(t)) + < \frac{\partial J(\boldsymbol{R}(t+1))}{\partial R_i(t)} > \\ &= \frac{\partial U(\boldsymbol{R}(t), \boldsymbol{u}(t))}{\partial R_i(t)} + \sum_j \frac{\partial U(\boldsymbol{R}, \boldsymbol{u})}{\partial u_j} \cdot \frac{\partial u_j(\boldsymbol{R}(t))}{\partial R_i(t)} \\ &\quad + \sum_j < \frac{\partial J(\boldsymbol{R}(t+1))}{\partial R_j(t+1)} \cdot \frac{\partial R_j(t+1)}{\partial R_i(t)} > \\ &\quad + \sum_{j,k} < \frac{\partial J(\boldsymbol{R}(t+1))}{\partial R_j(t+1)} \cdot \frac{\partial r_j(t+1)}{\partial u_k(t)} \cdot \frac{\partial u_k(t)}{\partial R_j(t)} >. \end{aligned} \tag{4}$$

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:

$$J'(\boldsymbol{R}(t), \boldsymbol{u}(t)) = U(\boldsymbol{R}(t), \boldsymbol{u}(t)) + < J'(\boldsymbol{R}(t+1), \boldsymbol{u}(\boldsymbol{R}(t+1)) > - U_0 \tag{5}$$

which, when differentiated, yields:

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

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 $\boldsymbol{u}(t)$ 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 $\boldsymbol{u}(t)$ as a function of $\boldsymbol{R}$ when differentiating J', then $\lambda^{(R)}$ would have been just the same as the λ 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, $\hat{J}(\boldsymbol{R}(t),W)$, which attempts to approximate the function $J(\boldsymbol{R}(t))$. Figure 13.1 illustrates the basic idea. The steps in HDP are as follows, starting from any actual or simulated value for $\boldsymbol{R}(t)$:

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

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

5. *Update* W in $\hat{J}(\boldsymbol{R}(t),W)$ based on *inputs* $\boldsymbol{R}(t)$ 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 $\boldsymbol{R}(t)$ and $\boldsymbol{R}(t + 1)$ 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 $\boldsymbol{R}(t)$ when dreaming is an outstanding research issue,

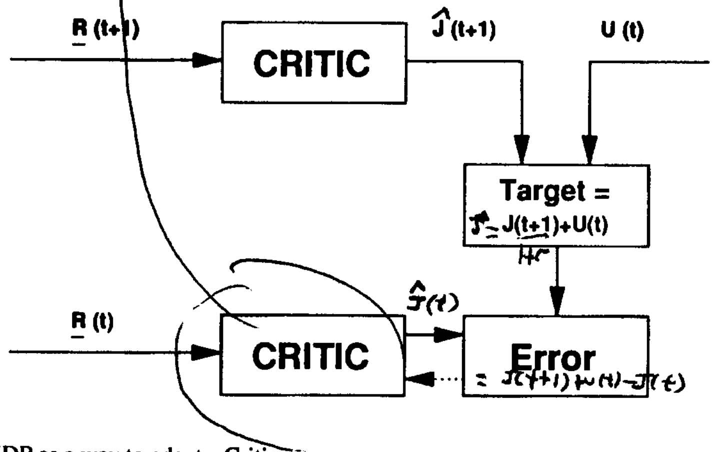

Figure 13.1 HDP as a way to adapt a Critic.

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 $\boldsymbol{R}(t)$ [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, $\hat{\lambda}(\boldsymbol{R}(t)$, which attempts to approximate the function $\lambda(\boldsymbol{R}(t))$ 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, λ^*, 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 $\boldsymbol{R}(t)$:

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

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

$$\boldsymbol{F\_u}(t) = F\_U_u(\boldsymbol{R}(t), \boldsymbol{u}(t)) + F\_f_u(\boldsymbol{R}(t), \boldsymbol{u}(t), \hat{\lambda}(t+1)) \tag{9}$$

$\underline{\lambda}^*(t)$ $$\lambda^*(t) = F\_f_R(\boldsymbol{R}(t), \boldsymbol{u}(t), \hat{\lambda}(t+1)) + F\_U_R(\boldsymbol{R}(t), \boldsymbol{u}(t)) + F\_A_R(\boldsymbol{R}(t), \boldsymbol{F\_u}(t)). \tag{10}$$

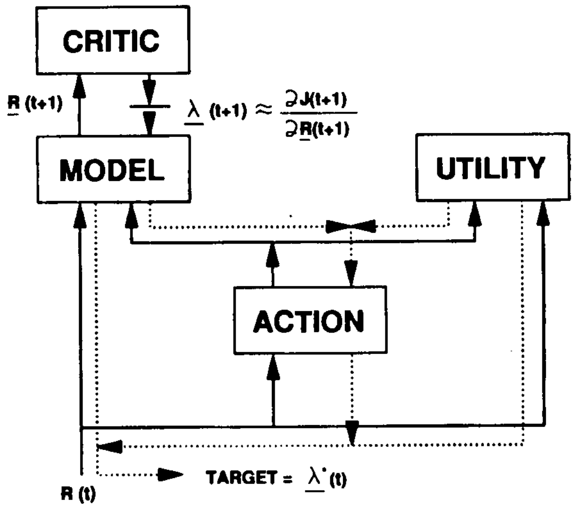

Figure 13.2 Calculating the targets λ^* in DHP.

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

In these equations, note that $\lambda(t+1)$ is input into the dual subroutine *in place of* $F\_\boldsymbol{R}(t+1)$. 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, $\hat{J}'(\boldsymbol{R}(t), \boldsymbol{u}(t), W)$, 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 $\boldsymbol{u}(t+1)$, they generate *all possible* action vectors $\boldsymbol{u}(t+1)$, calculate the resulting $\hat{J}'(t+1)$, and then pick the $\boldsymbol{u}(t+1)$ 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 $\boldsymbol{u}$ 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 $\boldsymbol{u}(t+1)$. (We certainly cannot *try* every $\boldsymbol{u}(t+1)$ 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 $\boldsymbol{u}(t+1)$ that are *not* optimal [9], but these are *not* used in calculating $\hat{J}'(t+1)$ when adapting the Critic network.

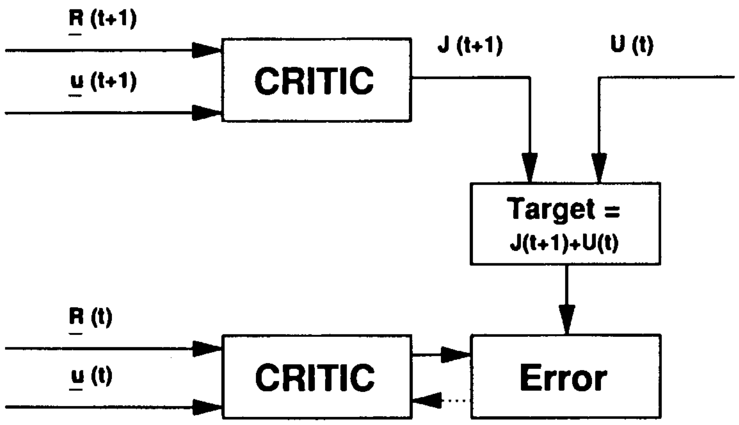

Figure 13.3 ADHDP as a way to adapt an action-dependent Critic.

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 $\boldsymbol{R}(t)$, the steps are:

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

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

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

2.6. Implementation of ADDHP

ADDHP adapts a Critic network with *two* outputs—$\lambda^{(R)}(\boldsymbol{R}(t), \boldsymbol{u}(t), W)$ and $\lambda^{(u)}(\boldsymbol{R}(t), \boldsymbol{u}(t), W)$—which try to approximate the two gradients of $J'(\boldsymbol{R}(t), \boldsymbol{u}(t))$. 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 $\boldsymbol{R}(t)$, the steps are:

1. Obtain $\boldsymbol{R}(t)$, $\boldsymbol{u}(t)$, and $\boldsymbol{R}(t+1)$ exactly as in HDP.
2. Calculate $\boldsymbol{u}(t+1) = \boldsymbol{A}(\boldsymbol{R}(t+1))$.
3. Calculate:

$$F\_\boldsymbol{R}(t+1) = \hat{\lambda}^{(R)}(\boldsymbol{R}(t+1), \boldsymbol{u}(t+1), W) + F\_A_R(\boldsymbol{R}(t+1), \hat{\lambda}^{(u)}(\boldsymbol{R}(t+1), \boldsymbol{u}(t+1), W)) \tag{12}$$

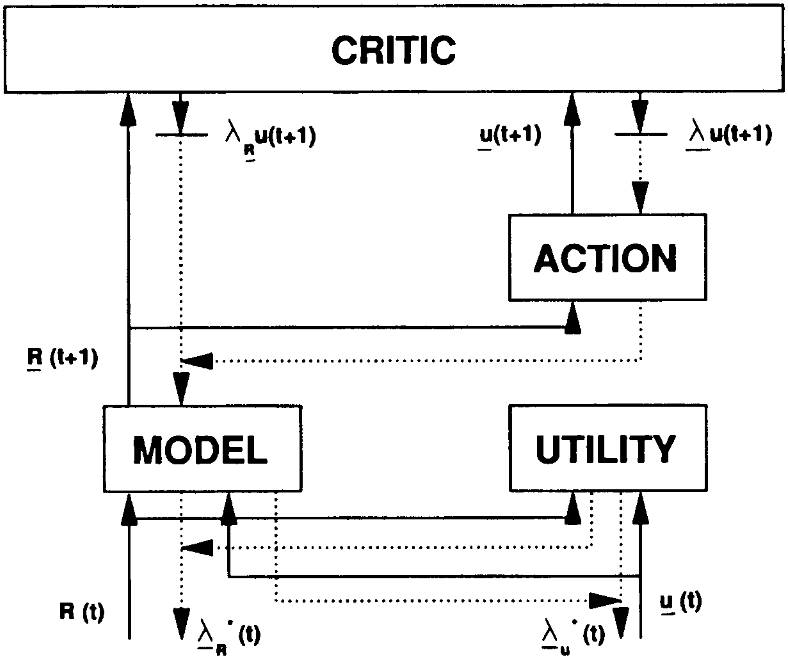
CRITIC
$\lambda_{\underline{R}}$u(t+1)
u(t+1)
$\underline{\lambda}$u(t+1)
ACTION
R (t+1)
MODEL
UTILITY
R (t)
$\underline{\lambda}_R^*$(t)
$\underline{\lambda}_u^*$(t)
u (t)

Figure 13.4 Calculating the targets λ_R^* and λ_u^* in ADDHP.

$$\lambda_R^*(t) = F\_f_R(R(t), u(t), F\_R(t+1)) + F\_U_R(R(t), u(t)) \tag{13}$$

$$\lambda_u^*(t) = F\_f_u(R(t), u(t), F\_R(t+1)) + F\_U_u(R(t), u(t)) \tag{14}$$

4. *Update* W in the Critic based on inputs $R(t)$ and $u(t)$ and targets $\lambda_R*(t)$ and $\lambda_u*(t)$.
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, $\boldsymbol{x}(t)$, 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:

$$U(t) \;=\; -\boldsymbol{x}(t)^T Q\boldsymbol{x}(t). \tag{15}$$

Suppose that the plant is governed by:

$$\boldsymbol{x}(t+1) \;=\; P\boldsymbol{x}(t)+R\boldsymbol{u}(t)+\boldsymbol{e}(t). \tag{16}$$

where:

$$<\boldsymbol{e}(t)\boldsymbol{e}(t)^T> \;=\; E. \tag{17}$$

Suppose that our Action network is:

$$\boldsymbol{u}(t) = A\boldsymbol{x}(t). \tag{18}$$

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:

$$P' = P + RA, \tag{19}$$

and observe that:

$$\boldsymbol{x}(t+1) = P'\boldsymbol{x}(t) + \boldsymbol{e}(t). \tag{20}$$

Let us define M as the matrix that solves the following equation:

$$M = P'MP'^T - Q. \tag{21}$$

(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:

$$J(\boldsymbol{x}) = \boldsymbol{x}^T M\boldsymbol{x}. \tag{22}$$

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:

$$J(\boldsymbol{x}(t)) = \boldsymbol{x}(t)^T M\boldsymbol{x}(t). \tag{23}$$

For the right-hand side, we get:

$$\begin{aligned} U(\boldsymbol{x}) + < J(\boldsymbol{x}(t+1)) > &= -\boldsymbol{x}^T Q\boldsymbol{x} \; + \; < (P'\boldsymbol{x}(t) + \boldsymbol{e}(t)^T M(P'\boldsymbol{x}(t) + \boldsymbol{e}(t)) > \\ &= -\boldsymbol{x}^T Q\boldsymbol{x} + \boldsymbol{x}(t)^T P'^T MP'\boldsymbol{x}(t) + < \boldsymbol{e}(t)^T M\boldsymbol{e}(t) >. \end{aligned}$$

(In this last calculation, I exploit the fact that $\boldsymbol{e}$ was assumed to be random, so that its cross-covariance with $\boldsymbol{x}$ is zero.) Because M satisfied equation 21, and because the distribution of $\boldsymbol{e}$ is a constant with respect to x, this tells us that:

$$U(\boldsymbol{x}) + < J(\boldsymbol{x}(t+1) > = \boldsymbol{x}^T(t)M\boldsymbol{x}(t) + U_0, \tag{24}$$

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:

$$J' = -x(t)^T Qx(t) + (Px(t) + Ru(t))^T M(Px(t) + Ru(t)). \tag{25}$$

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 $J(\boldsymbol{x})$ is $\boldsymbol{x}^T M\boldsymbol{x}$. The correct value for $\lambda(\boldsymbol{x})$ is simply the gradient of this. Thus, we start from:

$$\lambda(t+1) = 2M\boldsymbol{x}(t+1).$$

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 $\boldsymbol{x}(t)$, which yields an expected value of:

$$< P^T(2M\boldsymbol{x}(t+1)) > = 2P^T M(P+RA)\boldsymbol{x}(t).$$

For the second term, we simply use the gradient of $U(\boldsymbol{x}(t))$:

$$-2Q\boldsymbol{x}(t).$$

For the third term, we propagate $\lambda(t+1)$ through the model (equation 16) back to $\boldsymbol{u}(t)$, and then through the Action network, which yields an expected value of:

$$< A^T(R^T(2M\boldsymbol{x}(t+1)) > = 2A^T R^T M(P+RA)\boldsymbol{x}. \tag{26}$$

Adding all three terms together, we find that the expected value of our target vector will be:

$$< \lambda^*(t) > = 2(P+RA)^T M(P+RA)\boldsymbol{x}(t) - 2Q\boldsymbol{x}(t), \tag{27}$$

which is easily seen to equal $2M\boldsymbol{x}(t)$—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 $\boldsymbol{x}(t)$ 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 $\boldsymbol{u}$, at time $t + 1$, we start with:

$$\begin{aligned} <\lambda_{\boldsymbol{u}}(t+1)> &= <2R^TM(P\boldsymbol{x}(t+1)+R\boldsymbol{u}(t+1))> \\ &= 2R^TM(P+RA)<\boldsymbol{x}(t+1)> \;=\; 2R^TMP'<\boldsymbol{x}(t+1)> \end{aligned} \tag{28}$$

and likewise with $\boldsymbol{R}$:

$$<\lambda_{\boldsymbol{R}}(t+1)> \;=\; -2Q<\boldsymbol{x}(t+1)>+2P^TMP'<\boldsymbol{x}(t+1)>. \tag{29}$$

Following Figure 13.4, we first propagate $\lambda^{(u)}$ through the Action network, and add the result to $\lambda^{(R)}$ to get:

$$\begin{aligned} <\boldsymbol{F\_R}(t+1)> &= \lambda_{\boldsymbol{R}}(t+1)+A^T(\lambda_{\boldsymbol{u}}(t+1)) \\ &= -2Q<\boldsymbol{x}(t+1)>+2(P+RA)^TMP'<\boldsymbol{x}(t+1)> \\ &= -2Q<\boldsymbol{x}(t+1)>+2P'^TMP'<\boldsymbol{x}(t+1)>. \end{aligned}$$

By equation 37 this is simply:

$$<\boldsymbol{F\_R}(t+1)> \;=\; 2M\boldsymbol{x}(t+1)> \;=\; 2M(P\boldsymbol{x}(t)+R\boldsymbol{u}(t)).$$

(Notice that this complete derivative corresponds to the $\lambda(t+1)$ of DHP!) To calculate the target for $\lambda^{(u)(t)}$, we propagate *this* through the Model network back to $\boldsymbol{u}(t)$, and add in the gradient of utility with respect to $\boldsymbol{u}(t)$ (which happens to be zero in this example); the result is:

$$<\lambda_u^*(t)> \;=\; 2R^T M(P\boldsymbol{x}(t)+R\boldsymbol{u}(t)), \tag{30}$$

which corresponds to the correct value that we get from differentiating equation 25. In a similar way, the targets for $\lambda^{(R)}$ 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:

$$\begin{aligned} \boldsymbol{x}(t+1) &= P'\boldsymbol{x}(t)+\boldsymbol{e}(T) \\ U(\boldsymbol{x}) &= \boldsymbol{U}^T\boldsymbol{x} \\ J(\boldsymbol{x}) &= \boldsymbol{w}^T\boldsymbol{x}. \end{aligned} \tag{31}$$

(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:

$$<\boldsymbol{w}^{(n)}> \;=\; P'^T\boldsymbol{w}^{(n-1)}+\boldsymbol{U}. \tag{32}$$

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:

$$\lambda(\boldsymbol{x}) \;=\; \boldsymbol{w}.$$

Using calculations like those above, we see that a *single* time cycle is enough to yield *exactly*:

$$\lambda^* \;=\; P'^T\boldsymbol{w}+U. \tag{33}$$

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 $\lambda(\boldsymbol{R}(t), W)$, adapt $\hat{\lambda}(\boldsymbol{R}(t), \hat{\lambda}(\boldsymbol{R}(t-1)), W)$, *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, θ, 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 $U(\boldsymbol{R}(t), \boldsymbol{u}(t))$ with something like:

$$\hat{J}_{-}(\boldsymbol{R}(t_{-} + \theta)) + U_{-}(\boldsymbol{R}(t_{-})) - \hat{J}_{-}(\boldsymbol{R}(t_{-})) + U_{-}(\boldsymbol{R}(t), \boldsymbol{u}(t)), \tag{34}$$

where t_{-} is the most recent value of t divisible by θ (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 $\boldsymbol{u}(t)$ as the sum of $\boldsymbol{u}_{+}(t_{+} + \theta)$ and $\boldsymbol{u}_{-}(t)$. 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, $\boldsymbol{X}(t)$ and $\boldsymbol{Y}(t)$. As before, we wish to adapt a network that outputs a forecast of $\boldsymbol{Y}(t)$, based on our knowledge of $\boldsymbol{X}(t)$. 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, $\boldsymbol{X}(t+1)$, as a function f of the state at time t, the action vector $\boldsymbol{u}(t)$, 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 $\hat{\boldsymbol{X}}(t)$ 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):

$$\hat{\boldsymbol{X}}(t+1) = f_{\hat{X}}(\boldsymbol{X}(t), \hat{\boldsymbol{X}}(t), \boldsymbol{R}(t), \boldsymbol{u}(t), W) \tag{35}$$

$$\boldsymbol{R}(t+1) = f_R(X(t), \hat{X}(t), \boldsymbol{R}(t), \boldsymbol{u}(t), W) \tag{36}$$

$$Error = \sum_t E(t) = \sum_t L(X(t), \hat{X}(t), W). \tag{37}$$

As an example, *f* could be a two-stage system, consisting of an MLP which inputs $\boldsymbol{R}(t)$, $\boldsymbol{u}(t)$, and $\tilde{X}(t)$, and a simple filtering system which calculates $\tilde{X}(t)$ as a weighted average of $X(t)$ and $\hat{X}(t)$ (see Chapter 10). Intuitively, $\boldsymbol{R}(t)$ in this system represents memory from *before* time t. (The $\boldsymbol{R}$ in this system is slightly different from the $\boldsymbol{R}$ 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:

$$F\_\hat{X}(t) = F\_L_{\hat{X}}(X(t), \hat{X}(t), W) + F\_f_{\hat{X}}(X(t), \hat{X}(t), \boldsymbol{R}(t), \boldsymbol{u}(t), W, F\_\hat{X}(t+1), F\_\boldsymbol{R}(t+1)) \tag{38}$$

$$F\_\boldsymbol{R}(t) = F\_f_R(X(t), \hat{X}(t), \boldsymbol{R}(t), \boldsymbol{u}(t), W, F\_\hat{X}(t+1), F\_\boldsymbol{R}(t+1)) \tag{39}$$

$$F\_W = F\_W + F\_L_W(X(t), \hat{X}(t), W) + F\_f_W(X(t), \hat{X}(t), \boldsymbol{R}(t), \boldsymbol{u}(t), W, F\_\hat{X}(t+1), F\_\boldsymbol{R}(t+1)). \tag{40}$$

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 $F\_\boldsymbol{R}(t+1)$ 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 $\boldsymbol{R}(t)$ 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 $\boldsymbol{R}(t)$.

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, λ, which inputs $X(t)$, $X(t-1)$, $\boldsymbol{u}(t)$, $\boldsymbol{u}(t-1)$, etc., and outputs two quantities (λ_R and λ_X), which attempt to approximate:

$$\lambda_{\hat{X}}(X(t)\,...,\,W_\lambda) \approx F\_f_{\hat{X}}(X(t), \hat{X}(t), \boldsymbol{R}(t), \boldsymbol{u}(t), W_\lambda, \boldsymbol{F}\_\hat{X}(t+1), \boldsymbol{F}\_\boldsymbol{R}(t+1)) \tag{41}$$

$$\lambda_{\hat{R}}(X(t)\,...,\,W_\lambda) \approx F\_f_{\hat{R}}(X(t), \hat{X}(t), \boldsymbol{R}(t), \boldsymbol{u}(t), W_\lambda, \boldsymbol{F}\_\hat{X}(t+1), \boldsymbol{F}\_\boldsymbol{R}(t+1)). \tag{42}$$

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 $\boldsymbol{u}(t) = \boldsymbol{A}(\boldsymbol{X}(t), \boldsymbol{R}(t))$.
3. Calculate:

$$\begin{aligned}\lambda_{\hat{X}}(t) &= F\_L_{\hat{X}}(X(t), \hat{X}(t), W) + \lambda_{\hat{X}}(X(t), \,...,\, W_\lambda)\\ \lambda_{\boldsymbol{R}}(t) &= \lambda_R(X(t), \,...,\, W_\lambda).\end{aligned}$$

4. Set:

$$\begin{aligned}W = {} & W - LR(t) * (F\_L_W(X(t), \hat{X}(t), W)\\ & + F\_f_W(X(t-1), \hat{X}(t-1), \boldsymbol{R}(t-1), \boldsymbol{u}(t-1), W, \lambda_{\hat{X}}(t), \lambda_R(t))).\end{aligned}$$

   where $LR(t)$ is a learning rate (that can be adapted over time, etc., as in Chapter 3).
5. *Update* W_λ in λ to inputs $(X(t-1), X(t-2), \ldots)$ and targets:

$$\begin{aligned}\lambda^{*}\hat{X} &= F\_f_{\hat{X}}(X(t-1), \hat{X}(t-1), \boldsymbol{R}(t-1), \boldsymbol{u}(t-1), \boldsymbol{W}, \lambda_{\hat{X}}(t), \lambda_R(t))\\ \lambda^{*}\hat{R} &= F\_f_{\hat{R}}(X(t-1), \hat{X}(t-1), \boldsymbol{R}(t-1), \boldsymbol{u}(t-1), \boldsymbol{W}, \lambda_{\hat{X}}(t), \lambda_R(t)).\end{aligned}$$

6. Calculate and store for the next round:

$$\begin{aligned}\hat{X}(t+1) &= f_{\hat{X}}(X(t), \hat{X}(t), \boldsymbol{R}(t), \boldsymbol{u}(t), W)\\ \boldsymbol{R}(t+1) &= f_R(X(t), \hat{X}(t), \boldsymbol{R}(t), \boldsymbol{u}(t), W).\end{aligned}$$

As a practical matter, of course, there are many ways to try to "pipeline" these calculations. For example, one might merge f and λ into a single network; however, with λ 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 $\boldsymbol{R}(t+1)$ could be input instead of $\boldsymbol{R}(t)$ 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 $\boldsymbol{R}$ 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 λ_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:

$$X_i(t+1) = \hat{X}_i(t+1) + \sigma_i e_i(t+1), \tag{43}$$

where e_i represents random noise of unit variance, and σ_i^2 represents the variance of the error in predicting X_i. (In other words, σ_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 σ_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 $\mathbf{R}(t+1)$, 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* $\mathbf{R}(t)$ as the starting point in the same procedure, and to *simulate* $\mathbf{R}(t+1)$ using equation 43, and then to treat the simulated $\mathbf{R}(t+1)$ 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 $< ee^T >$ is diagonal, and that the randomness in the *observed* variables always follows a normal distribution. Conventional control theory usually allows for *any* matrix $< ee^T >$, 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:

$$X_i(t+1) = \sigma_i^X e_i^X(t+1) + D_i(R(t+1), \textit{other information}(t)) \tag{44}$$

$$R_i(t+1) = \sigma_i^R e_i^R(t+1) + P_i(information(t)), \qquad (45)$$

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 (σ_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 though 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 $\hat{\boldsymbol{R}}(t)$. Then we can call on the Encoder network, which inputs $\boldsymbol{X}(t)$, along with any or all information available to the system from time $t-1$. (This could, in principle, include the prediction $\hat{\boldsymbol{R}}(t)$, which does not incorporate real information from time t.) The output of the Encoder network is a vector, $\bar{\boldsymbol{R}}$, which is a kind of best guess of the true value of ***R***. The next step is to generate simulated values of ***R***, $\boldsymbol{R}'$, by adding random numbers to each component $\bar{\boldsymbol{R}}_i$. Finally, the Decoder network inputs these *simulated* values, $\boldsymbol{R}'$, *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 σ_i^R and σ_i^X.

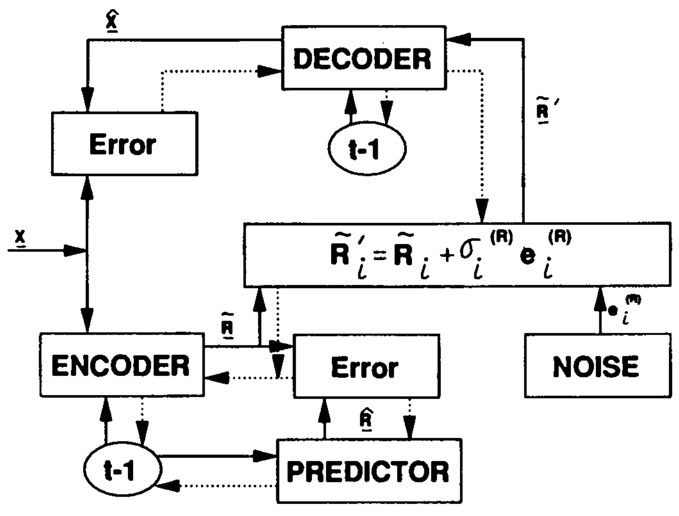

Figure 13.5 The Stochastic Encoder/Decoder/Predictor.

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 σ_i^X is straightforward. (For example, σ_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 σ_i^R is more difficult. Conceptually, we try to adapt *all* parts of the network *together* so as to minimize:

$$E = \sum_i (X_i - \hat{X}_i(R'(x, \sigma^R, e^R)))^2/(\sigma_i^X)^2 + \sum_j (\hat{R}_j - R_j(x))^2 \tag{46}$$

$$+ \sum_i \log (\sigma_i^X)^2 + \sum_j ((\sigma_j^R)^2 - \log (\sigma_j^R)^2).$$

In performing this minimization, it is critical that we account for the effect of σ^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 σ_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 $\boldsymbol{R}_j$ as:

$$F\_\boldsymbol{R}_j = 2(\boldsymbol{R}_j - \hat{\boldsymbol{R}}_j) + F\_\hat{\boldsymbol{R}}'_j \tag{47}$$

where the first term results from differentiating the R-prediction error in equation 46 with respect to $\boldsymbol{R}_i$, and the second term represents the derivative of the X-prediction error, calculated by backpropagation through the Decoder network back to $\boldsymbol{R}_j'$. Note that equation 46 requires us to divide the X_i prediction error by $(\sigma_i^X)^2$, 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 $F\_\boldsymbol{R}$ 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 $\boldsymbol{R}$ 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 $\boldsymbol{R}$, 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:

$$\frac{\partial E}{\partial(\sigma_j^R)^2} = F\_\boldsymbol{R}_j' * e_j + \frac{\partial}{\partial(\sigma_j^R)^2}((\sigma_j^R)^2 - \log(\sigma_j^R)^2),$$

where the right-hand term tells us to make σ_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 $\boldsymbol{R}(t + 1)$ 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 $\boldsymbol{R}(t + 1)$, starting from data at time t. First one uses the Predictor network to generate $\hat{\boldsymbol{R}}(t + 1)$. To simulate $\boldsymbol{R}_i(t + 1)$, one adds $\hat{\boldsymbol{R}}_i(t + 1)$ to a random number, whose variance is set to the observed average error in predicting $\boldsymbol{R}_i$ from $\boldsymbol{R}_i$. Next one generates a new set of random numbers, e^R, and generates $\boldsymbol{R}'(t + 1)$ as shown in Figure 13.5. Finally, one uses the Decoder network to generate $\hat{\boldsymbol{X}}$ and a third set of random numbers to simulate $\boldsymbol{X}$. In backpropagating through this structure, one simply proceeds as if the random numbers had been external inputs to the system.

To use actual observations, $\boldsymbol{X}(t + 1)$, one plugs these observations into the Encoder network, which then outputs the actual values of $\boldsymbol{R}(t + 1)$. One uses these actual values in simulating $\boldsymbol{R}'(t + 1)$, which is then plugged into the Decoder network to generate $\hat{\boldsymbol{X}}(t + 1)$. We still use random numbers in simulating $\mathbf{R}'$, but we use actual errors, in effect, to infer what the random numbers "actually were" in generating $\mathbf{R}$ and $\mathbf{X}$. 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 $\mathbf{R}(t+1)$. 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 σ^R 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:

$$\mathbf{x}(t) = A\mathbf{R}(t) + \mathbf{e}(t), \tag{48}$$

where:

$$<\mathbf{R}\mathbf{R}^T> = I \tag{49}$$

and:

$$< e_i(t)e_j(t) > \triangleq S^X_{ij} = I_{ij}(\sigma^X_i)^2, \tag{50}$$

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 $\mathbf{R}$, 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:

$$\sum_j ((\sigma^R_j)^2 - \log(\sigma^R_j)^2)$$

by:

$$Tr(S^R) - \log \det S^R,$$

where S^R is the covariance *matrix* used to generate the random vector $\boldsymbol{e}^R$. (Note that my notation for $\boldsymbol{e}^R$ here is slightly different from what it is in Figure 13.5.) I will assume that the "neural network" takes the form:

$$\boldsymbol{R} = \beta \boldsymbol{x} \tag{51}$$

$$\boldsymbol{R}' = \beta \boldsymbol{x} + \boldsymbol{e}^R \tag{52}$$

$$\hat{\boldsymbol{x}} = \alpha \boldsymbol{R}' = \alpha(\beta \boldsymbol{x} + \boldsymbol{e}^R) \tag{53}$$

$$< \boldsymbol{e}^R (\boldsymbol{e}^R)^T > = S^R. \tag{54}$$

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 σ_i^x 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:

$$\alpha = A. \tag{55}$$

A correct Encoder should output the best estimate of $\boldsymbol{R}(t)$ given knowledge of $\boldsymbol{x}(t)$, based on knowledge of equation 48. We can calculate this best estimate, $\beta \boldsymbol{x}(t)$, simply by regressing $\boldsymbol{R}$ on $\boldsymbol{x}$, using classical methods [15] that are known to minimize square error:

$$\beta = < \boldsymbol{R}\boldsymbol{x}^T > (< \boldsymbol{x}\boldsymbol{x}^T >)^{-1}.$$

To work this out, note that our original linear model implies:

$$\begin{aligned} < \boldsymbol{x}\boldsymbol{x}^T > &= < (A\boldsymbol{R} + \boldsymbol{e})(A\boldsymbol{R} + \boldsymbol{e})^T > \\ &= < A R R^T A^T > + < \boldsymbol{e}\boldsymbol{e}^T > = AA^T + S^X. \end{aligned} \tag{56}$$

(Note that our assumption that e is random implies $<\boldsymbol{ex}> = <\boldsymbol{eR}> = 0$.) Likewise:

$$<\boldsymbol{Rx}^T> = <\boldsymbol{R}(A\boldsymbol{R}+\boldsymbol{e})^T> = <\boldsymbol{RR}^TA^T> = A^T.$$

Substituting this into our formula for β, we deduce that the correct value is:

$$\beta = A^T(AA^T+S^x)^{-1}. \tag{57}$$

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:

$$\beta = A^T(AA^T+S^x)^{-1}. \tag{58}$$

We also assume that S^R has been adapted to the following value:

$$S^R = I - A^T(AA^T+S^x)^{-1}A. \tag{59}$$

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 $\boldsymbol{R}'(t)$. From classical statistics [15], we know what this results in, in the linear case:

$$\alpha = <\boldsymbol{xR}'^T>(<\boldsymbol{R}'\boldsymbol{R}'^T>)^{-1}.$$

Clearly this will lead to a unique solution for α. We want to verify that it will lead to the correct solution; in other words, we want to verify:

$$\begin{aligned} A &= <\boldsymbol{xR}'^T>(<\boldsymbol{R}'\boldsymbol{R}'^T>)^{-1} \\ &= <\boldsymbol{x}(\beta\boldsymbol{x}+\boldsymbol{e}^R)^T>(<(\beta\boldsymbol{x}+\boldsymbol{e}^R)(\beta\boldsymbol{x}+\boldsymbol{e}^R)^T>)^{-1} \\ &= <\boldsymbol{xx}^T>\beta^T(\beta<\boldsymbol{xx}^T>\beta^T+S^R)^{-1}. \end{aligned} \tag{60}$$

This is equivalent to:

$$A(\beta<\boldsymbol{xx}^T>\beta^T+S^R) = <\boldsymbol{xx}^T>\beta^T.$$

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:

$$AS^R + AA^T(AA^T + S^x)^{-1}A = A. \tag{61}$$

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 β. We are minimizing the following effective error function with respect to β:

$$\begin{aligned} E_{eff} &= <(x-\hat{x})^T S^{x-1}(x-\hat{x})> + <\tilde{R}^T R> \\ &= <(x-AR')^T S^{x^{-1}}(x-AR')> + <\tilde{R}^T R> \\ &= <(x-A(\beta x+e^R))^T(S^x)^{-1}(x-A(\beta x+e^R))> + <(\beta x)^T(\beta x)> \\ &= <x^T(I-A\beta)^T(S^x)^{-1}(I-A\beta)x> + <x^T\beta^T\beta x> + \textit{terms not affected by } \beta. \end{aligned}$$

Thus, we want to minimize the following matrix trace with respect to β:

$$Tr(<xx^T>(\,(I-A\beta)^T(S^x)^{-1}(I-A\beta)+\beta^T\beta\,)). \tag{62}$$

We can simplify this minimization problem slightly by defining matrices:

$$\begin{aligned} Z &= <xx^T> \\ C &= (S^x)^{-1}A \\ Q &= A^T(S^x)^{-1}A + I. \end{aligned}$$

Using this notation, we are trying to minimize the terms in equation 62 that depend on β:

$$E_{eff} = -2\sum_{i,j,k} Z_{ij}C_{jk}\beta_{ki} + \sum_{i,j,k,l} Z_{ij}\,\beta^T_{jk}\,Q_{kl}\,\beta_{li}. \tag{63}$$

Differentiating equation 63 with respect to every matrix element β_{ij}, and collecting terms carefully, we arrive at the following condition for a minimum:

$$0 = -2C^TZ + Q\beta Z + Q\beta Z$$

$$\therefore\ C^T = Q\beta.$$

Going back to our definitions, this yields a solution for β of:

$$\beta = Q^{-1}C^T = (A^T(S^x)^{-1}A + I)^{-1}A^T(S^x)^{-1.} \tag{64}$$

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:

$$A^T(AA^T + S^x)^{-1} = (A^T(S^x)^{-1}A + I)^{-1}A^T(S^x)^{-1.} \tag{65}$$

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:

$$(A^T(S^x)^{-1}A + I)A^T = A^T(S^x)^{-1}(AA^T + S^x), \tag{66}$$

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 $<E>$ affected by S^R:

$$E_{eff} = <(x - \hat{x})^T(S^x)^{-1}(x - \hat{x})> + Tr(S^R) - \log\det S^R$$

$$= <(e^R)^T A^T(S^x)^{-1}Ae^R> + Tr(S^R) - \log\det S^R + \textit{terms not affected by } S^R.$$

This reduces to an effective error function to be minimized of:

$$E_{eff} = Tr(S^R M) - \log\det S^R, \tag{67}$$

where I have defined:

$$M = A^T(S^x)^{-1}A + I. \tag{68}$$

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:

$$S^R = M^{-1}. \tag{69}$$

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:

$$(A^T(S^x)^{-1}A+I)^{-1} = I-A^T(AA^T+S^x)^{-1}A. \tag{70}$$

To verify this, we simply multiply the right-hand side of equation 70 by $I+A^T(S^x)^{-1}A$, and verify that we arrive at the identity matrix. When we do this, we get:

$$\begin{aligned}
&(I+A^T(S^x)^{-1}A)(I-A^T(AA^T+S^x)^{-1}A)\\
&= I+A^T(S^x)^{-1}A-A^T(AA^T+S^x)^{-1}A-A^T(S^x)^{-1}AA^T(AA^T+S^x)^{-1}A\\
&= I+A^T(S^x)^{-1}A-A^T(I+(S^x)^{-1}AA^T)(AA^T+S^x)^{-1}A\\
&= I+A^T(S^x)^{-1}A-A^T(S^x)^{-1}(S^x+AA^T)(AA^T+S^x)^{-1}A\\
&= I+A^T(S^x)^{-1}A-A^T(S^x)^{-1}A = I,
\end{aligned}$$

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 α, β, 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 $\boldsymbol{R}$. Such a rotation does not affect the matrix $<\boldsymbol{RR}^{\mathrm{T}}> = I$. Since A^TA is a positive definite real matrix, we know that we can always arrive at an equivalent form of the model that diagonalizes A^TA, 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, $\mathbf{R}$ (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, $\hat{\mathbf{R}}$, 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 $\hat{\mathbf{R}}$. 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 $\mathbf{R}$) *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 Markhov 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.