Neural Networks for Control, Edited by W. T. Miller, R. S. Sutton, and P. J. Werbos, Cambridge, MA: The MIT Press, 1990, Chapter 3.

3 A Menu of Designs for Reinforcement Learning Over Time

Paul J. Werbos

3.1 Introduction and Overview

This chapter will provide a menu of designs for artificial neural networks which learn to maximize a measure of utility, reinforcement or performance over time. By defining utility as the negative of cost, we can use these same designs to minimize a measure of cost.

Maximization problems of this sort arise in a variety of contexts. For example, when we want a robot arm to follow a desired trajectory, and we want to minimize the energy it uses, we can translate this into a problem of cost minimization: we simply define the measure of cost as the sum of two terms, one representing the deviation between the actual trajectory and the desired trajectory, and the other representing energy expenditure. Jordan (1989) has shown that this approach actually works in simulation tests. Kawato, in chapter 9, takes a similar approach to trajectory planning, as do Nguyen and Widrow in chapter 12. Psychologists have studied the idea of learning based on reinforcement signals for decades. Many problems in industry can be stated as problems in long-term profit maximization.

In designing neural networks to solve these problems, the biggest challenge is to account for the link between *present* actions and *future* consequences. There are two basic ways to meet this challenge. One way is to build an explicit model of the external environment which the neural net is trying to control, and use backpropagation through time (BTT) to calculate the derivatives of future utility with respect to present actions. (All references to BTT in this chapter will refer to this particular way of using backpropagation. See Werbos (1990b) for a tutorial on backpropagation through time in more general applications). The other way is to adapt a "critic" network, a special network which outputs an *estimate* of the total future utility which will arise from present situations or actions. This chapter will focus on the adaptive critic approach.

This book presents these two approaches as ways of adapting artificial neural networks. However, both approaches can be applied just as easily to adapting *any* network of differentiable functions. They can both be viewed as general methods in control theory. For example, Werbos (1989a) used BTT to maximize profits in the natural gas industry, as part of the official 1988 forecasts of the Energy Information Administration; the model of the environment in that case was a simple economic model of that industry. From the viewpoint of control theory, BTT is

the same as the first-order calculus of variations (Bryson and Ho 1969), modified *only* by the use of a more efficient, parallel algorithm to calculate the derivatives (Lagrange multipliers). Likewise, the adaptive critic can be derived as a way of *approximating* dynamic programming; dynamic programming, in turn, is the *only* exact and efficient method available to control motors or muscles over time, so as to maximize a utility function in a noisy, nonlinear environment, without making highly specialized assumptions about the nature of that environment. Section 3.7 will describe how to implement the ideas in this chapter in the general situation, not limited to neural networks as such.

BTT is easy to use and exact, but it has no provision for handling noise or error in one's model of the plant, and it is not suitable for real-time learning as in biological systems (Werbos 1988b). Thus the very first journal article mentioning backpropagation (Werbos 1977) focused on its use in a *subordinate* role, as an adjunct to the adaptive critic approach. In this approach, backpropagation may be used from cell to cell at *a given time*, but not backwards through time. Many people believe that backpropagation is not biologically plausible even in that limited role; however, arguments can be made for its plausibility (e.g., Werbos 1988a, 1988b), especially in light of new evidence that the cytoskeleton *inside* of mammalian cells can transmit information at speeds on the order of millimeters per millisecond (Zhu and Skalak 1988).

This chapter will focus in depth on the adaptive critic approach, which is a large and important subject in its own right. Other chapters and papers cited above already describe BTT (chapter 4 also describes some variants of BTT). For background information on all these methods, see chapter 1 and chapter 2. For concise mathematical definitions and examples, see section 3.7.

Adaptive critic designs come in many shapes and forms. The simplest and best-known designs have been criticized for their inability to handle very large control problems. This chapter will argue that these criticisms are legitimate, but that we can overcome them by using more complex adaptive critic designs. Many applications do not require such complexity, but in the long-term—to duplicate or explain the capabilities of the brain—we will need to work with it. This chapter will begin by presenting a simple, generic design for an adaptive critic system, and then—section by section—show how additional features can be added, to arrive at additional capabilities.

Section 3.2 will present a simple design, made up of only two neural networks—a critic network and an action network. Designs like this have worked well in real, physical experiments (not just simulations);

however, the examples I know of are proprietary, confidential, or so recent that I have only heard oral presentations. (For example, John Mayhew of Sheffield University in England has spoken about applications to controlling a simple autonomous vehicle.)

Section 3.3 will describe two minor limitations of the simple design, related to the possibility of divergence and the impact of unobserved variables. It will describe how to overcome these limitations, by going back to the literature on dynamic programming, and upgrading the design to reflect that literature.

Sections 3.4 and 3.5 will focus on the deeper, more difficult problem of how to handle a *large number* of control variables (e.g., many motors or muscles to control) or sensor variables. Certainly the human brain is capable of handling millions of control variables, and there are many engineering applications where a number of motors must be controlled in tandem. Section 3.4 will describe a way of coping with these problems, in adapting the action network, and will draw some parallels with the human brain. Section 3.5 will go further, by describing alternative ways to adapt the critic network itself.

Section 3.6 will discuss a few topics for future research, such as the problem of high-speed motor control and the problem of extending the effective planning horizon. Section 3.7 will give equations and examples to assist in implementation.

3.2 A Simple Two-Component Adaptive Critic Design

3.2.1 What the Design Tries to Do

This section will describe the design of a neural network system to solve the problem of reinforcement learning. Before describing that system, I will first describe the problem which it tries to solve.

The problem of reinforcement learning has very deep roots in the literature of psychology. Sutton (1984) has traced this problem back to Marvin Minsky, in the early 1960s and earlier.

Figure 3.1 illustrates the problem. Imagine a little man (or computer) sitting in front of a row of levels labeled u_1 through u_n. Beside him is a big meter (which looks like a thermometer in the cartoon), labeled U. The meter gives a measure of how well the man is doing. His job is to control the levers so as to make U as large as possible. The blinking lights, labeled X_1 through X_m, are simply a source of information which the man can use in deciding which levers to pull and when to pull them. In principle, the man could start out with no knowledge of the lights

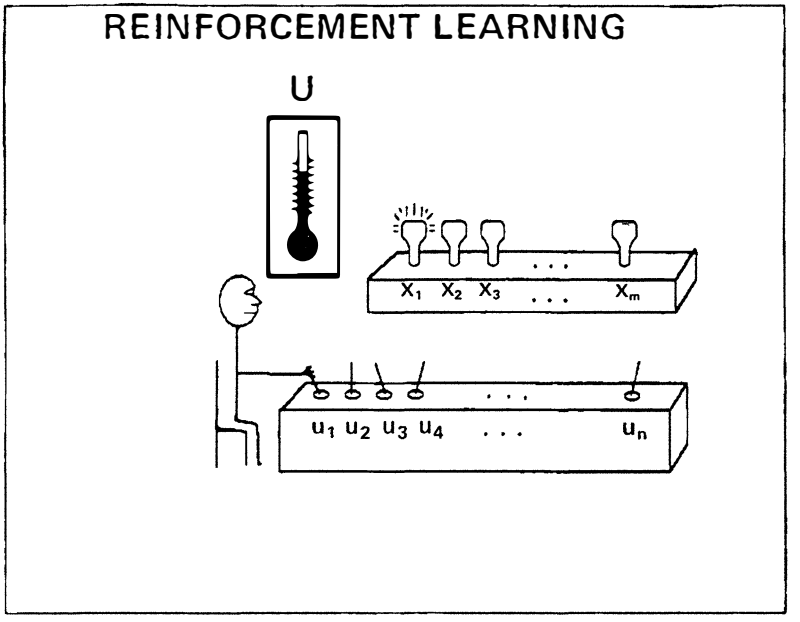

Figure 3.1
Cartoon image of reinforcement learning over time.

or the meter; he may have to learn everything from scratch, allowing for the possibility of nonlinear fluctuations and noise in the external environment.

This notation is similar to that used in decision and control theory. The collection of variables u_1 through u_n form a vector, $\underline{u}$, called the control vector (Bryson and Ho 1969). The observation variables X_1 through X_m form a vector $\underline{X}$, similar to the observation vector of control theory. The variable U—called "reinforcement" in psychology—plays the same role as the utility function U in the theory of cardinal utility developed by John Von Neumann and popularized by Howard Raiffa. (U may also be used to represent a "performance index" or a "cost function" or a "profit function").

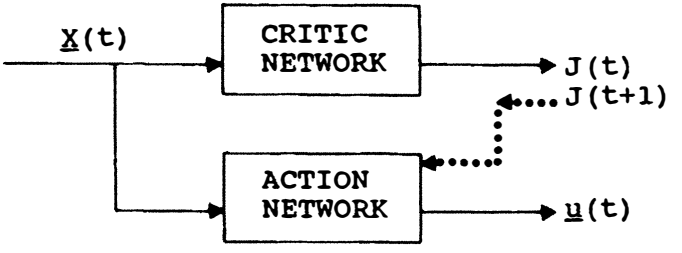

Figure 3.2
A simple adaptive critic system.

Williams (1988) has discussed earlier formulations of reinforcement learning, in which the actions $\underline{u}(t)$ are chosen to maximize $U(t)$, as if later times did not matter. This chapter will focus entirely on the problem of reinforcement learning *over time*, in which the goal is to maximize the long-term expected value of $U(t)$. We will always try to account for the effect of current actions on the state of the world at later times, either explicitly or implicitly.

3.2.2 Overview of the Two-Component Design

Figure 3.2 shows one way to build an adaptive critic system, taken (with modification) from Sutton (1984). Two neural networks are adapted over time—an action network and a critic network. The action network outputs the actual control signals, $\underline{u}(t)$, while the critic network guides how the action network is adapted. The critic network inputs a description of the state of the world ($\underline{X}(t)$) and outputs a single number, $J(t)$, which is an evaluation of how well the action network is doing in creating a "good" situation. (The letter J stands for "Judgement," and is also used by Bryson and Ho (1969) and Raiffa (1968).) Then, in each time t, the action network is "rewarded" or "punished," based on what kind of situation it produces; in other words, actions $\underline{u}(t)$ are "rewarded" if they lead to good results (larger $J(t+1)$) and "punished" if they lead to bad results (smaller $J(t+1)$).

To translate this psychological intuition into a working design, we need two things:

1. A learning rule which states how the weights in the action network change in response to reward and punishment ($J(t+1)$);

2. A learning rule which states how the weights in the critic network change as a result of experience.

To adapt the action network, Barto, Sutton and Anderson (1983) adjusted the weights in response to correlations between various variables calculated in the network and the variable J. I will not discuss the details of their method, in part because they are already well known, and in part because section 3.4 will discuss some alternatives.

The remainder of this section will describe a simple form of heuristic dynamic programming (HDP), a method for adapting the critic network proposed in Werbos (1977). This chapter will not describe the better-known method of temporal differences used by Barto, Sutton and Anderson to adapt their critic networks; however, Werbos (1990a) argues that HDP may be viewed as a generalization of that method. Grossberg and his collaborators have also developed critic-like networks, which are beyond the scope of this chapter.

3.2.3 Description of HDP

As noted above, the critic network yields an output, J, which is a function of its current inputs, $\underline{X}$, and of its weights, $\underline{w}$. Let us write this function as $J(\underline{X}, \underline{w})$.

Intuitively, we want to find weights $\underline{w}$ which make $J(\underline{X}(t), \underline{w})$ an accurate assessment of "how good" $\underline{X}(t)$ is. (Section 3.3 will provide a more rigorous basis for this idea.) We want to know how good $\underline{X}(t)$ is, relative to the problem we started with—maximizing the expected value of U across all future times. Ideally, then, we would want $J(\underline{X}(t), \underline{w})$ to be an estimate of:

$$U(t) + U(t+1) + ... + U(\infty) \tag{3.1}$$

assuming that this sum converges. (Section 3.3 describes what to do if not.)

How can we train the critic network to provide such an estimate? HDP, like backpropagation, can be implemented in a variety of ways. For example, it can be implemented through real-time learning (where the weights are updated after *each pattern* is analyzed), or it can be implemented through batch learning (where the weights are updated all at once after a big pass through all the patterns). For simplicity, I will consider the case of batch learning here.

In order to simplify our future discussion, it will help to remember that $U(t)$ is available at time t as *one of the observed* inputs to the system. Thus we can assume that it is available as one of the components of the vector $\underline{X}$; in other words, $X_k(t) = U(t)$ or some k. This lets us write U as a function of X, $U(\underline{X}) = X_k$. Let us also assume that we are given a time series of vectors, $\underline{X}(t)$, for t going from 1 to T. Once again, our goal is to adapt the weights $\underline{w}$ in the critic network.

In HDP, we use some sort of supervised learning method to adapt the critic. *Any* supervised learning method will do, *so long as* we use the right inputs and targets. For example, anyone familiar with basic backpropagation should be able to fill in the rest of the details, *when* the inputs to the net and the targets are fully specified, for the training set.

In the simplest form of HDP, we carry out several passes through the training set. We start with an initial set of weights $\underline{w}^{(0)}$; in pass number n, we derive a new set of weights $\underline{w}^{(n)}$. We keep going through the training set, over and over, until the weights settle down, i.e. until $\underline{w}^{(n+1)} = \underline{w}^{(n)}$.

On each pass, our training set consists of $T - 1$ pairs of inputs and target, for $t = 1$ through $T - 1$. (There is only one target for each pair, because the critic network has only one output, J.) The inputs for time t are simply the vector $\underline{X}(t)$. The target for time t, within pass number n, is:

$$J(\underline{X}(t+1), \underline{w}^{(n-1)}) + U(\underline{X}(t)) \tag{3.2}$$

In other words, before we *begin* the adaptation of the weights in pass number n, we have to plug in $\underline{X}(t+1)$ into the critic network, using the old weights, in order to calculate the *target* for each time t. The targets are then *fixed* throughout pass number n. Then, in the adaptation phase of pass number n, we update the weights to try to reach the targets. We could do this in an exhaustive way, by searching the entire weight space, or by using a supervised learning method (like Kohonen's pseudo-inverse method) which converges in a single pass; alternatively, we could simply do a single pass of backpropagation.

Theory tells us that we should simply throw out the case where $t = T$, when we are trying to control an infinite, continuous process, because equation 3.2 is not well defined in that case (since $\underline{X}(T+1)$ is unknown). In many practical applications, however, our set of observations $\underline{X}(1)$ through $\underline{X}(T)$ is actually made up of several *strings* of observations, where each string represents a different *experiment* on the system to be

controlled. In those applications, $\underline{X}(T)$ usually represents the *completion* of the last experiment; thus the case $t = T$ should be added to the data set, with a target of $U(\underline{X}(t))$. In fact, the target for the final time t of *any* such *string* is simply $U(\underline{X}(t))$. In some applications, we may even want to use a different utility function, U, for the final time, to measure how well the experiment is completed; this can be done quite easily when using HDP.

Werbos (1990a) has shown that this simple form of HDP converges to exactly the right weights, for a simple class of linear problems. More precisely, it converges whenever the external environment is a linear system, governed by a matrix equation with multivariate normal noise, when the Critic network has the appropriate form (which is simply linear in this case), and when the supervised learning method is itself statistically consistent.

3.3 HDP and Dynamic Programming

3.3.1 Background

Figure 3.3 illustrates the basic trick used in *dynamic programming*, which is one of the fundamental tools used in control theory. Dynamic programming is discussed in standard textbooks like Bryson and Ho (1969) and Gale (1979), but the extension of dynamic programming developed by Howard (1960) is most relevant here.

Dynamic programming requires as its input a utility function U and a model of the external environment or plant, which I denote as "$\underline{F}$". Dynamic programming *produces*, as its major output, *another* function, J^*, which I like to call a secondary or *strategic* utility function. The key insight in dynamic programming is that you can maximize the expected value of U, in the *long term*, *over time*, simply by maximizing the function J^* in the immediate future. Whenever you know the function J^* and the model $\underline{F}$, it is a simple problem in function maximization to pick the actions which maximize J^*. The function J of section 3.2 may be viewed as an *approximation* to the function J^*.

Why should we train our Action network to maximize an approximation like J instead of the exact function J^*? Clearly it is better to use the exact function, when this is possible. But the computational cost of finding J^* grows exponentially with the number of variables in the problem. To cope with complicated problems in the general case, we really have only one choice; we have to approximate dynamic programming, by using a model or network to estimate J^* function or its derivatives.

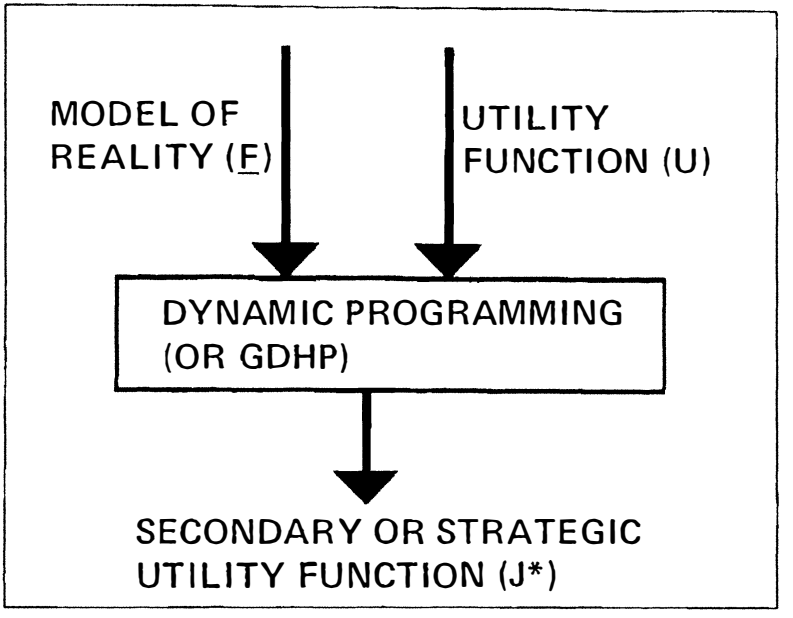

Figure 3.3
What dynamic programming requires and produces.

The derivatives of $J^*(\underline{X}, \underline{w})$ with respect to the variables X_k form a vector $\underline{\lambda}^*$; the Lagrange multipliers used in the calculus of variations are an example of this vector. In fact, all approximation methods—including neural net methods—for "solving" the reinforcement learning problem over time in a general environment may be viewed as general approximations to dynamic programming; in all cases which I am familiar with, they do include specific functions which try to approximate J^* or $\underline{\lambda}^*$ in a general way. Examples may be found in numerous fields, including artificial intelligence, physics, economics, and schools of psychology ranging from the behaviorist to the avowed mystic (Werbos 1986).

By looking at dynamic programming more closely, we will see two features—involving two new terms, $\underline{R}$ and U_0—to enhance the design of section 3.2.

3.3.2 Mathematics

Dynamic programming assumes the availability of a model of the external environment or plant which we may write as:

$$\underline{R}(t+1) = \underline{F}(\underline{R}(t), \underline{u}(t), \underline{noise}) \tag{3.3}$$

where $\underline{u}(t)$ refers to the vector of actions at time t, noise is a vector of random numbers, and $\underline{R}$ is a vector describing the current state of Reality—the plant or the environment. For mnemonic purposes, it helps to recall that "R" stands for "recurrent," "representation" and "reconstruction"—all of which apply to some extent.

At this point, some readers may believe that the shift from $\underline{X}$ to $\underline{R}$ is an oversight or an unnecessary inconvenience. However, this shift is truly fundamental to the validity of the method. *If* equation 3.3 were an accurate description of reality with $\underline{R} = \underline{X}$, then the shift would indeed be unnecessary; however, this is usually not the case in practical applications and is certainly not the case in human psychology. For example, when a person sees a ball roll under a chair, he does not act as if the ball has vanished from reality. Likewise, in control theory, the state vector and the vector of observables are usually quite distinct.

How can we build a neural network to represent the function $\underline{F}$? A simple, popular approach is to build a network which inputs $\underline{X}(t)$, which uses $\underline{X}(t+1)$ as its targets, and which includes recurrent hidden units. In that case, the vector $\underline{R}(t)$ would consist of the outputs of the hidden units and all the components of $\underline{X}(t)$. Unfortunately, this approach has a number of weaknesses, involving its robustness over long time periods, its representation of noise, and problems involving real-time convergence. Werbos (1987a) offers a few suggestions for overcoming these weaknesses, based on extensive empirical work, but more research is needed. The first stage of Kawato's cascade method is an example of what Werbos (1977) calls the "pure robust method"; it is a good first step, but very far from the whole story. Difficulties in this area are arguably the most serious obstacle now facing us in neurocontrol.

In any event, it is safe to treat $\underline{X}$ as *part* of the vector $\underline{R}$, since what we observe is part of reality. For this reason, we may treat U as a function of $\underline{R}$, $U(\underline{R})$. Henceforth, I will also define:

$$\lambda_i^*(t) = \frac{\partial}{\partial R_i} J^*(\underline{R}(t)) \tag{3.4}$$

Howard has proven that the function J^* can be found by solving a modified form of the Bellman equation.

$$J^*(\underline{R}) = Max < J^*(\underline{F}(\underline{R}, \underline{u}, \underline{noise})) > U(\underline{R}) - U_0 \quad (3.5)$$

where U_0 is an intercept term used to prevent drift off to infinity, and where the angle brackets denote the average or expected value. (Notational standards vary greatly here; angle brackets are standard in physics and some branches of statistics, because they minimize confusion in long calculations.) Howard's equation was slightly more complicated, in that he allows U to depend on $\underline{u}$ as well as $\underline{R}$; that feature is of little value here, and would lead to unnecessary confusion.

Comparing equation 3.5 with equation 3.2, it is clear that "U_0" is an additional complication. *If* there is no possibility of drift off to infinity (i.e., if the expected value of equation 3.1 converges), then U_0 is unnecessary. Such convergence can result either from uncertainty about the future which grows with time (Werbos 1990a) or from discount rates (as in Barto, Sutton and Anderson 1983). When equation 3.1 does not converge, one can modify HDP by estimating U_0 explicitly. For example, one can define $J(\underline{R}) + U_0$ as the output of the network, and keep the targets as in equation 3.1; one can adapt U_0 as one adapts any other weight in the network. Werbos (1979) talks at length about problems related to the existence of U_0; fortunately, these problems seem far removed from the usual neural net applications.

Equation 3.5 assumes the existence of a model, $\underline{F}$, while section 2 did not. The point here is that Section 3.2—following Barto, Sutton and Anderson (1983)—used the world itself as a model of itself. Instead of using a model, $\underline{F}$, and *simulating* $\underline{F}(\underline{R}, \underline{u}, \underline{noise})$ to get a simulated $\underline{R}(t+1)$, it used reality itself to generate $\underline{R}(t+1)$. This should yield the correct expectation values, after enough experience, but there is one catch: we may need to develop a model $\underline{F}$ anyway, to help us develop a representation vector $\underline{R}$, in applications where that is important.

Howard has proven many theorems which are relevant to HDP. One of them may be roughly translated as follows. Suppose that we have a critic network $J(\underline{R}, \underline{w})$ and an action network u(R,w') so rich that they can represent *any* functions $J^*(\underline{R})$ and $\underline{u}^*(\underline{R})$ by choosing the appropriate weights. Suppose that we adapt these networks by going back and forth between two steps:

1. Update $\underline{w}$ such that $J(\underline{R}(t), \underline{w}^{(n)})$ *equals* $J(< \underline{R}(t+1) >, \underline{w}^{(n-1)}) + U(\underline{R}(t)) - U_0$, for *all* possible vectors $\underline{R}(t)$, where the expectation value of $\underline{R}(t+1)$ refers to the expectation assuming that $\underline{u}(\underline{R}, \underline{w}'^{(n-1)})$ is used to control system actions;

2. Update $\underline{w}'$ such that $\underline{u}(\underline{R}(t), \underline{w}'^{(n)})$ maximizes $< J(\underline{R}(t+1), \underline{w}^{(n)} >$.

Then the weights $\underline{w}$ and $\underline{w}'$ will converge to give the *optimal* strategy of action. J will converge to J^*.

3.4 Alternative Ways to Figure 3.2 in Adapting the Action Network

3.4.1 The Challenge of Large Problems

The design shown in figure 3.2 has been criticized by Guez (1987) and others, because it provides such little feedback to guide the action network. There is only a single number, $J(t+1)$, used to evaluate all aspects of action. This is very different from the usual situation in supervised learning, where separate error measures are available to each of the output cells. Here, there is no indication of which output cell should have had a different output, or of which way the output should have been different (smaller versus larger). For this reason, when there are many outputs to be controlled, one can expect much slower learning than one would get with supervised learning. The problem here is like the problem of a student trying to figure out what to study when his professor only gives him an overall grade (like J), instead of giving him feedback targeted to individual questions on the exam.

These intuitive arguments can also be expressed in statistical terms. When there are a very large number of variables (like all the weights in a network) correlated against a single dependent variable (like J), one faces a problem called multicollinearity (Wonnacott and Wonnacott 1977). With such multicollinearity, it becomes nearly impossible to estimate the correct weights, unless one has an astronomically large data base. The point is that the system learns slowly, when there are many output variables, for reasons which have nothing to do with the numerical *slowness* reported with some versions of backpropagation. The system is slow in the sense that the estimated weights, *after* numerical convergence, still converge very slowly to the true weights as the size of the database, T, goes to infinity. (Multicollinearity is an issue in supervised learning as well (Werbos 1987a), but the assumption of a sparse structure tends to keep it from getting out of hand.)

3.4.2 A Simple Way of Meeting the Challenge

Figure 3.4 illustrates an alternative to figure 3.2, based on the use of backpropagation, proposed in Werbos (1977).

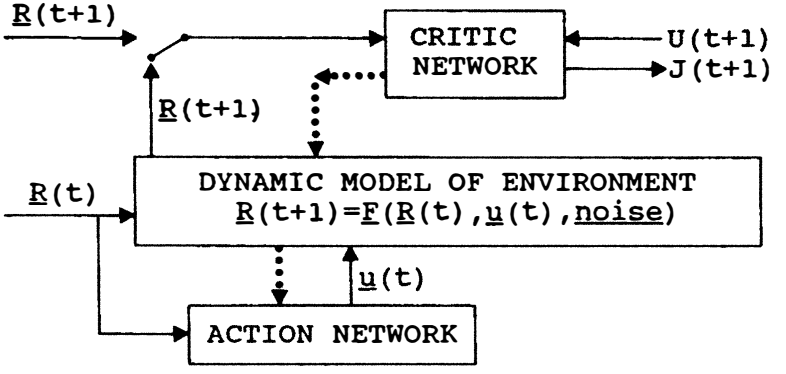

Figure 3.4
Backpropagated adaptive critic (BAC). (Note how j is not used directly, but its derivatives—the dotted lines—are.)

Figure 3.4 is based on the idea of following equation 3.5 more exactly. After all, our goal is to find a vector $\underline{u}$ which maximizes $J(\underline{F}(\underline{R}, \underline{u}))$. Figure 3.2 is a lot like plugging in values of u and looking only at the values of the function $J(\underline{F}(\underline{R}, \underline{u}))$. But with backpropagation, we can calculate the derivatives of $J(\underline{F}(\underline{R}, \underline{u}))$ with respect to all the components of $\underline{u}$, in a single pass through the system. In numerical analysis, at least, we can find the maximum of a function much faster if we exploit the gradient information, instead of looking only at the value of the function. (When the gradient is used with steepest ascent, convergence may be very slow, but Werbos (1988a) describes other ways of using the gradient information.)

The basic idea here is very straightforward. The action network can be represented by a function, $\underline{u}(\underline{R}, \underline{w}')$. Our goal is to adapt the weights w_i'. To do this, we use backpropagation to calculate the derivatives of $J(\underline{F}(\underline{R}, \underline{u}(\underline{R}, \underline{w}')))$ with respect to the w_i'. Then we adjust the weights in response to the derivatives, as in conventional backpropagation. In this case, we are *backpropagating* through the critic to the model ($\underline{F}$) and then to the action network, as if the three networks formed one large feedforward network. Information is passed backwards (the dotted lines) from network to network in figure 3.4, but it is not propagated backwards in time. (See chapter 10 for more details.)

This approach may be viewed as a synthesis of the best features of figure 3.2 and of backpropagation through time. As with backpropagation through time, we have *specific* feedback to *specific* components of $\underline{u}$, rather than one gross evaluation. This feedback accounts for the cause-and-effect information embedded in the network $\underline{F}$. If a machine or person never uses knowledge of cause and effect to guide his choice of actions, then there are limits to how well he can cope with a complex environment.

3.4.3 Procedures for Handling Noise

Williams (1988) has pointed out that we have to be very careful in propagating derivatives through a network (like F in figure 3.4) which has noise attached to it. Therefore, I will be very specific about how to do this.

There are two different ways to handle the noise vector in the $\underline{F}$ network—the simulation approach and the imputed-noise approach. I will describe both of these approaches for the case of batch learning, as in Section 3.2.

Assume that we have a set of vectors $\underline{R}(t)$ for $t = 1$ through T. Assume that the model network $\underline{F}$ and the critic network J will be held constant. Our goal for now is to adapt the action network $\underline{u}(\underline{R}, \underline{w}')$.

In the simulation approach, we begin by simulating the noise vectors. For each value of t and each component of the vector noise, we pick a value at random, based on the probability distribution for that component. This procedure will yield vectors $\underline{noise}(t)$ for all t, which we hold constant thereafter. Then on each pass through the data, calculate the derivatives of J with respect to a weight w_i' as:

$$\frac{\partial}{\partial w_i'} \sum_t J(\underline{F}(\underline{R}(t), \underline{u}(\underline{R}(t), \underline{w}'), \underline{noise}(t))),$$

treating both $\underline{\text{noise}}$ and $\underline{\text{R}}(\text{t})$ as constants. Many variations are possible, such as the simulation of multiple noise vectors for each $\underline{R}(t)$, resimulation after each major pass, and comparison tests across different simulations.

Many readers will ask where we find a probability distribution for each component of $\underline{noise}$. In fact, this probability distribution is really *part* of our model of the plant. It may come from our substantive knowledge of the plant, or it may come from a neural network adapted to represent the plant; as discussed in section 3.3, it is not a simple issue how we should adapt such a network.

One way of developing such a model of the plant is to build a network which inputs $\underline{R}(t)$ and $\underline{u}(t)$, and aims at $\underline{R}(t+1)$ as the targets for its output. We can then assume the following stochastic model of the plant:

$$\underline{R}(t+1) = \hat{\underline{R}}(\underline{R}(t), \underline{u}(t)) + \underline{noise}, \tag{3.6}$$

where the only form of noise is error in the forecast $\underline{R}(t+1)$. We can assume that the noise for each component is normally distributed, and we can estimate the variance by simply looking at the variance of $R_i - \hat{R}_i$ across the training set. (For high performance, one can modify this very slightly, to assume that the noise acts on the activation level of the output cells.)

When our model of the plant looks like equation 3.6, we have a very simple alternative to the simulation approach. For each pair of vectors $\underline{R}(t)$ and $\underline{R}(t+1)$ in our data, we can simply plug $\underline{R}(t)$ into the forecasting network $\hat{\underline{R}}$, and impute:

$$\underline{noise} = \underline{R}(t+1) - \hat{\underline{R}}(\underline{R}(t), \underline{u}(t))$$

In fact, this procedure turns out to be very simple in practice. It amounts to plugging in the *actual* $\underline{R}(t+1)$ into the critic for the first part of the backpropagation (through the critic), but backpropagating straight on through to the forecasting network and the action network as if there had been no noise. We don't even have to estimate the variance of the noise, because we never really use it for anything.

The approach of section 3.2 is really an example of the imputed noise approach. However, the simulation approach could also be used with HDP as well. Furthermore, it is also possible to simulate the vectors $\underline{R}(t)$ themselves, when using the simulation method. The imputed noise approach has the advantage of staying close to observed data, and therefore being more robust and reliable. The simulation approaches (like "dreaming") have the advantage of allowing the system to prepare for situations it can predict but has never experienced; for example, if humans were not capable of dreaming, we would not be capable of embarking on projects like space programs or building new devices which have never existed in the past. It is impossible to map out a complex decision space without including *some* provision—either dreaming or actual exploration—to ensure that all regions in the space are considered.

3.4.4 Beyond the Design of Figure 3.4

Figure 3.4 has certain advantages over figure 3.2, which have been discussed. However, it has disadvantages as well. For example, its validity

depends on the validity of the model of the plant, $\underline{F}$. Figure 3.2 may be slower but more reliable at times, since it sticks to observed data only. If there is a sudden, unexpected painful event, a system like Figure 3.2 may be able to respond immediately, by changing its weights, instead of waiting for a change in the model of the plant which can explain the situation. (On the other hand, models of the plant like Kawato's may be a better guide to action than the empirical data, at times, because they can filter out confusing short-term fluctuations.)

These relative strengths and weaknesses are similar in flavor to the strengths and weaknesses of backpropagation versus associative memory in conventional supervised learning (Werbos 1987a, 1988a). Once again, one would like to find a blend which combines the best of both methods. This blend is really needed only in the most sophisticated applications, where it is important to combine the advantages of both.

One possible blend has emerged from discussion between Barto and myself this past year. In essence, one can start with HDP, as in section 3.2, but also adapt an auxiliary network $J'(\underline{u}, \underline{R})$ which tries to predict the error, $J(\underline{R}(t))$ minus the targets in equation 3.2. Then one can backpropagate through the critic and the model network back to $\underline{u}$, as in figure 3.4, but also backpropagate through the J' network directly to $\underline{u}$. If J' is adapted by a high-speed supervised learning method (like associative memory), then this will provide an immediate reward or punishment to weights which lead to unexpected good or bad results. There are many refinements possible here, and other alternatives as well. This approach can also be extended quite easily to the methods of section 3.5. A similar structure, based in different mathematical objectives, has worked well in simulations by Lukes et al. (1990).

3.4.5 Biological Parallels

The human brain is not a homogeneous mass of identical neurons, but it is not a preprogrammed hierarchy of cells with fixed specific tasks either. During the workshop which led to this book, James Houk stated that the brain is actually quite flexible and modular; it is made up of five or so major subsystems, each containing a high degree of flexibility (albeit with different cell types) within it. From a functional point of view, standard texts like Nauta and Feirtag (1986) suggest that the five major adaptive subsystems are: (1) the limbic system; (2) the basal ganglia; (3) the thalamus/cerebral-cortex system; (4) the brain stem; (5) the cerebellum.

Figure 3.4 shows only three major subsystems. Werbos (1987b) cites extensive physiological evidence consistent with the idea that the limbic system is like a critic network, the corticothalamic system like a model network (F), and the brain stem like an action network. But this leaves out the basal ganglia, including the nucleus basalis, which recent research points to as very important. The connections of the basal ganglia are more like figure 3.2 than figure 3.4, suggesting that the brain might represent a kind of blend of these architectures. This blend may be very different in quality from the blend discussed in this section; if so, it is all the more important to study these differences in detail, because they may point to new opportunities on the engineering side. The cerebellum is well known to be responsible for smoothing out high-speed motor control, to be discussed in section 3.6.

3.5 Alternatives to HDP in Adapting the Critic Network

Section 3.4 began with a discussion of the challenge of large problems, as it affects the adaptation of the action network. The exact same problem also affects the adaptation of the critic network. If we use HDP, as described in section 3.2, then we still have that problem, even if we use the design of figure 3.4 to adapt the action network. HDP still only uses *one* piece of information—the target for J as given in equation 3.1—to adapt the critic network.

3.5.1 Utility Functions

One useful step in solving this problem is to exploit our knowledge (if any) of the utility function, $U(\underline{R})$. Section 3.2 showed how the reinforcement learning problem can be represented as a special case of utility maximization, of maximizing some function $U(\underline{R})$. However, the methods of this chapter can all be applied to an arbitrary differentiable utility function. If we know apriori how to connect our performance measure to more concrete variables (like the position of a robot arm), then we can avoid wasting time as our system relearns this information. The examples of backpropagation through time cited in section 3.1 all take advantage of this.

Klopf (1982) has placed great stress on the importance of these effects. He defines the problem of drive reinforcement learning as the problem of maximizing $U(\underline{R})$ when $U(\underline{R})$ is known to obey:

$$U(\underline{R}) = \sum_i V_i R_i, \tag{3.7}$$

where the V_i are treated as fixed weights. This is more plausible biologically than the situation shown in figure 3.1, because it allows for the fact that primary reinforcement comes from a variety of cells, not just one "master cell."

It is very straightforward to extend HDP to the case of drive reinforcement learning. We need to define our critic as a network with multiple outputs, J_{i+}, one for each component of the vector $\underline{V}$. Conceptually, we may define:

$$J = \sum_i J_i, \tag{3.8}$$

but we never really need to calculate J. We can still use supervised learning, as in section 3.2, but we now need targets for *each* output J_i. The target for J_i is simply:

$$J_i(\underline{R}(t+1), \underline{w}^{(n-1)}) + V_i R_i(t),$$

with allowance for U_{i0} if convergence is a problem, as discussed in Section 3.2. (Alternatively, we could weight the sum in equation 3.8 by V_i, and eliminate the V_i in equation 3.9.) To adapt the action network, we can backpropagate through the critic just as easily as we could before; we begin by noting that the derivative of J with respect to each J_i is simply 1, and proceed backwards in the usual way.

This does not tell us how to exploit our knowledge of $U(\underline{R})$ if U is a nonlinear function or network. To do that, we must go beyond HDP to more advanced methods.

3.5.2 Dual Heuristic Programming (DHP)

In examining figure 3.4, Barto has asked whether we really need the critic network at all here. All we are really getting from the critic network are the derivatives, the dotted lines feeding back from the critic to the model of the plant ($\underline{F}$). If we could estimate the derivatives of J^* with respect to $\underline{R}(t+1)$ directly, then we would not need the critic at all. This is basically what DHP tries to do.

Strictly speaking, DHP does not eliminate the critic. It replaces the critic with a new network, whose output is an estimate of the derivatives of J^* with respect to $\underline{R}$. We can think of this new network as an alternative type of critic. The derivatives of J^* with respect to $\underline{R}$ are simply the vector $\underline{\lambda}^*$, defined in equation 3.4. We may say that DHP adapts

a λ-type critic, rather than a J-type critic. When we adapt the action network, we use this new critic to calculate $\underline{\lambda}$, and then propagate these derivatives back through the model of the plant and the action network just as we would if $\underline{\lambda}$ had come out of backpropagation.

How can we adapt a λ-type critic? First let us consider the theory behind DHP. DHP is based on *differentiating* equation 3.5 on both sides (and suppressing the expectation operator and maximization, as in Section 3.2) to get:

$$\frac{\partial J^*}{\partial R_i(t)} = \sum_j \frac{\partial J^*}{\partial R_j(t+1)} \frac{\partial F_j}{\partial R_i(t)} + \frac{\partial U}{\partial R_i(t)} \tag{3.9}$$

Note that the unpleasant U_0 term has dropped out. Recalling our definition of $\underline{\lambda}^*$, and defining the rightmost term as $V_i(\underline{R})$, we get:

$$\lambda_i^*(t) = \sum_j \lambda_j^*(t+1) \frac{\partial F_j}{\partial R_i(t)} + V_i(\underline{R}) \tag{3.10}$$

Using DHP, we no longer need to assume the availability of $U(\underline{R})$; instead, we assume the availability of the reinforcement vector $\underline{V}$, which—unlike the $\underline{V}$ of the previous section—truly varies over time, as a function of the situation. The summation in the middle of the equation is simply an application of the chain rule, a calculation of derivatives; to calculate such derivatives efficiently in practice, we can simply use backpropagation.

All of this theory leads to the following procedure. With DHP, as in HDP, we adapt a critic network. Our critic now has multiple outputs, λ_i. The input to the critic is $\underline{R}(t)$, exactly as in HDP. We can use any supervised learning method to adapt this network, to make the outputs match the targets. We have to iterate through many passes, as in HDP, even if we use a one-pass supervised learning method in each pass.

The only difference with HDP is in where we get the targets. Instead of using equation 3.2, we now try to use the right-hand side of equation 3.11. In order to use the right-hand side of equation 3.11, we have to use backpropagation *as a way of calculating* the summation term in that equation. In other words, DHP forces us to use backpropagation as a way to obtain the targets, not as a way to adapt the network to match the targets. (Backpropagation can be used in both places, of course, if we so desire, as in the Appendix.)

More precisely, in each pass n, we assume that the weights $\underline{w}^{(n-1)}$ are available (as in HDP). Our first step (as in HDP) is to calculate the targets for each time t. We do this by

1. Inserting $\underline{R}(t+1)$ as the input into the λ-critic, using the old weights $\underline{w}^{(n-1)}$ in that network. The output will be $\underline{\lambda}(t-1)$.

2. Backpropagating these derivatives (i.e., the components of $\underline{\lambda}(t+1)$) through the model of the plant ($\underline{F}$), all the way back to the $R_i(t)$ variables input to that model. This yields an estimate of the derivatives of J^* with respect to each of the $R_i(t)$.

3. Set the target for $\lambda_i(t)$ to be $V_i(\underline{R}(t))$ *plus* this new estimate of the derivative of J^* with respect to $R_i(t)$.

The next step is to adapt the weights so as to make the outputs closer to the targets, exactly as in HDP.

Just like the BAC design of figure 3.4, DHP takes full advantage of the cause-and-effect information embedded in the model of the plant ($\underline{F}$). It focuses the computational effort on estimating the slope of J^*, which is usually more important to making good decisions than is the absolute level of J^*.

3.5.3 Globalized DHP (GDHP)

As in section 3.4, there are numerous possibilities for blends between the two basic methods. GDHP is one such blend.

HDP has the advantage of *coherence*. Because there is only one J function, there is one consistent evaluation of how well one is doing. DHP, however, is not guaranteed to be internally consistent; the derivative of $\lambda_i(\underline{R})$ with respect to R_j ought to equal the derivative of $\lambda_j(\underline{R})$ with respect to R_i, but it may not come out that way in our approximation. Ideally, one would want to know the absolute level of J—for use in making big decisions—while also learning about the slope in fine detail.

GDHP makes this possible. In GDHP, we adapt a J-type critic, as in HDP, but we try to minimize the error in equation 3.10, as in DHP. (As in HDP, we treat the weights on the right-hand side as constants, to avoid the problems discussed in Werbos (1990a).) In fact, we could even expand this error measure by adding it to the error function implicit in HDP.

Unfortunately, the only way I know to minimize such an error measure is to use backpropagation to adapt the critic network. In using backpropagation, we need to calculate the derivatives of error with respect to the weights in the critic network; however, the error measure *itself* contains derivatives, so that we need to calculate *second* derivatives. The details of this are given in the Appendix.

Better methods may be found to achieve the same objectives, but I am not aware of any such at present. The best I can imagine at present is the adaptation of further networks to approximate some of these calculations.

3.6 Some Topics for Further Research

3.6.1 The Problem of "Vision" (Effective Long-Term Foresight)

Even in Howard's dynamic programming, the current estimate of J^* in any iteration will tend to represent the future H periods into the future (i.e., it will really tell you how to maximize the sum of $U(t)$ through $U(t+H)$, not $U(t)$ through $U(\infty)$). On each full application of the Bellman equation, H grows to $H+1$. But if the cycle time for calculation is, say, a fifth of a second, and the unit time period is a fifth of a second as well, it would take years to build up a time horizon of years *at best*; realistic inefficiencies, and a lack of a complete update in neural networks, could lead to a time horizon of only a few days after years of learning.

If adaptive critics were adapted by backpropagation, one could replace steepest descent by an accelerator method which could work far faster—at the risk of instability (a risk which may have its biological counterpart). Also, most accelerator methods do not allow for this kind of "moving target" problem, where parameter changes themselves change the target.

Werbos (1987b, 1990a) mentions a few ideas on how to cope with this problem, but it is a wide-open area. It is also a difficult area, and less than essential to near-term engineering applications. Still, it may be very important to human intelligence.

3.6.2 The Problem of High-Speed Motor Coordination (Cerebellum)

GDHP takes a long time to go through a cycle of calculations, because there are so many calculations to go through. Werbos (1987b) compared it in detail with literature on the human cerebral cortex and limbic system, which are also relatively slow and relatively coherent. The brain needs a faster system to smooth out actions—the cerebellum.

How can we link a *primary* neurocontrol system, based on something like GDHP, with a *secondary* optimization system which is less

integrated, less statistically efficient (i.e., learns slowly), but much faster in operation (smaller cycle time)? How could we build an artificial cerebellum? Perhaps we might use something like $\lambda_i(t+1) - \lambda_i(t)$—as estimated by the primary system—to be used as V_i in a subordinate, fast HDP system without a model. The multivariate richness of the $\underline{\lambda}(t+1) - \underline{\lambda}(t)$ vector would partly overcome the usual slowness of HDP. Or perhaps a very simple associative model could be used with DHP, or other inputs could be used in HDP with a goal of simply smoothing motion. The biological studies discussed by Kawato in chapter 9 suggest very strongly that the human cerebellum does indeed minimize a cost measure (torque change) over time. In general, the interface between multiple sets of neurocontrollers—including humans as well—will be important to many practical applications.

3.6.3 The Need For Tests

This is almost certainly the most critical research area for now. All the trade-offs discussed above seem fairly clear from the mathematics, but concrete tests are needed, across a wide spectrum of problems, to clarify and communicate the nature of these tradeoffs in practice. Likewise, a creative approach to "making things work"—by diagnostic analysis and modification as necessary—is vital to solving realistic problems using any of the methods in this book. Few things in this field are likely to work out perfectly the very first time they are tried, when they are implemented in the most trivial way; in chapter 7 Shanno cites examples of the same phenomenon in numerical analysis and function minimization, which present very similar challenges. Diagnostic analysis—drawing on a wide range of disciplines as well as immersion in the behavior of concrete examples—will be essential, as will mathematical analysis of simplified problems which abstract the essence of more realistic ones.

3.7 Equations and Code For Implementation

Many engineers would find it difficult to understand the ideas above without seeing a few equations. Unfortunately, the equations will look very different, depending on the type of network to be used (neural net versus econometric model versus fuzzy logic net versus fluid dynamics code versus ...), the type of learning schedule, the type of computer (sequential versus parallel versus dedicated), etc. This section will describe the key details, as they might look in a sequential computer, applied to real-time learning, using my approach to handling noise

and basic backpropagation with steepest ascent to handle all supervised learning problems. This is not the best way to go, but it should at least clarify the basic ideas.

The flowcharts in this chapter exploit a modular way of thinking about systems design. To preserve this modularity, I will present the key calculations in terms of subroutines which are then linked together to form a system. The result will look a lot like real code, but there has been no effort to optimize its efficiency or carry out numerical testing/debugging.

3.7.1 Preliminaries

A functional network may be defined as a subroutine $NET(\underline{X};\underline{w};\underline{x};\underline{Y})$, which inputs arrays $\underline{X}$ and $\underline{w}$ and outputs arrays $\underline{x}$ and $\underline{Y}$, and performs the following calculations internally:

$$x_i = X_i, \quad i \leq m$$

$$x_i = f_i(x_{i-1}, ..., x_2, x_1, \underline{w}), \quad i = m+1, ..., N$$

$$Y_i = x_{i+N}, \quad i = 1, ..., n$$

where m, n and N are constants built into the subroutine, and the f_i are twice-differentiable functions also built into the subroutine. $\underline{Y}$ is the "real" output of the network, but the entire array $\underline{x}$ is sometimes needed.

Artificial neural networks are a special case of functional networks, where f_{n+2j+1} for all j calculates a weighted sum of the outputs of earlier neurons, and f_{n+2j+2} calculates $1/(1+exp(-x_{n+2j+1}))$—the output of the $(j+1)$st neuron. It is possible to handle networks where f_i is allowed to use x_{i+1}, x_{i+2}, etc., as arguments, in addition to the arguments shown here; however, this requires special methods (Werbos 1988b).

For any functional network, we can construct a dual subroutine, F_NET($F\_\underline{Y};\underline{x};F\_\underline{w};F\_\underline{X};F\_\underline{x}$). The inputs to this subroutine are the arrays $F\_\underline{Y}$ and $\underline{x}$. The main outputs are $F\_\underline{w}$ and $F\_\underline{X}$. ($F\_\underline{x}$ is usually just scratch space.) The dual has the following key property: if $F\_\underline{Y}$ represents the gradient of some quantity L with respect to Y, and Y is the output of the subroutine NET, then $F\_\underline{w}$ and $F\_\underline{X}$ will be the gradients of L with respect to the weights $\underline{w}$ and the inputs $\underline{X}$, respectively. (Werbos (1989a) gives a more rigorous definition of what this means.)

The dual can be coded as follows, based on backpropagation:

$$F\_x_{i+N} = F\_Y_i$$

$$F\_x_i = \sum_{j=i+1}^{N+n} F\_x_j \frac{\partial f_j}{\partial x_i}(\underline{x}, \underline{w}), \quad i = N, ...1$$

$$F\_w_i = \sum_{j=m+1}^{N+n} f\_x_j \frac{\partial f_j}{\partial w_i}(\underline{x}, \underline{w}), \quad \text{all weights}$$

$$F\_X_i = F\_x_i, \quad i \leq m$$

Finally, for GDHP, we need to pay special attention to networks NET which have only a single output. For the critic network, we need to construct a *doubly dual* subroutine, G_F_NET($\underline{x}$; $F\_\underline{x}$; $Weight\_\underline{X}$; $G\_\underline{w}$), whose first three arguments are inputs and whose last argument is an output. If the output of the subroutine NET is a single scalar, J, then we want to calculate:

$$G\_w_i = \sum_j Weight\_X_j \frac{\partial^2 J}{\partial X_j \partial w_i}$$

To calculate these derivatives efficiently, we may use the tricks in Werbos (1988a), which lead to the following equations:

$$G\_F\_x_i = Weight\_x_i + \sum_{j=1}^{i-1} G\_F\_x_j \frac{\partial f_i}{\partial x_j}, \quad i = 1, ..., N$$

(where $Weight\_x_i = 0$ for $i > n$)

$$G\_x_{N+1} = 0$$

$$G\_x_i = \sum_{j=i+1}^{N+1} \left(G\_x_j \frac{\partial f_j}{\partial x_i} + F\_x_j \sum_{k=1}^{j-1} G\_F\_x_k \frac{\partial^2 f_j}{\partial x_i \partial x_k} \right), i = N, ..., m+1$$

$$G\_w_i = \sum_{j=1}^{N+1} (G\_x_j \frac{\partial f_j}{\partial w_i} + F\_x_j \sum_{k=1}^{j-1} G\_F\_x_k \frac{\partial^2 f_j}{\partial w_i \partial x_k})$$

For relatively sparse networks (and neural networks especially), these second derivatives tend to be very sparse as well, and the summations turn out to be relatively easy (see Werbos 1988a).

3.7.2 Implementing Section 3.4

To implement the BAC method of figure 3.4, we require three subroutines or networks, whose arguments are ordered by the conventions just stated:

CRITIC($\underline{R}; \underline{w}; \underline{x}; J$)
MODEL($\underline{R}(t), \underline{u}(t), \underline{noise}; \underline{w}''; \underline{x}''; \underline{R}(t+1)$)
ACTION($\underline{R}(t); \underline{w}'; \underline{x}'; \underline{u}(t)$)

Our goal is to adapt $\underline{w}$ and $\underline{w}'$; we assume that $\underline{w}''$ is already known. Note that the input array to MODEL consists of the concatenation of three vectors. As in a real computer program, the order of arguments—not their names—is what controls the process.

When we use the simulation approach to handling noise, and we use real-time learning, we can assume that we start from a vector R describing reality. We next simulate a noise vector noise, and must then adjust the weights. To adjust the action network according to figure 3.4, we calculate:

CALL ACTION($\underline{R}; \underline{w}'; \underline{x}'; \underline{u}$)
CALL MODEL($\underline{R}, \underline{u}, \underline{noise}; \underline{w}''; \underline{x}''; \underline{R2}$)
CALL CRITIC($\underline{R2}; \underline{w}; \underline{x}; J$)
CALL F_CRITIC($1; \underline{x}; scratch; F\_R2; scratch$)
CALL F_MODEL($F\_R2; \underline{x}''; scratch; scratch, F\_u, scratch; scratch$)
CALL F_ACTION($F\_u; \underline{x}'; F\_w'; scratch; scratch$)
$w' = w' + learning\_rate * F\_w'$

where "scratch" refers to scratch space (i.e., unused outputs).

3.7.3 Dual Heuristic Programming (DHP)

To implement DHP under the same conditions, we change our CRITIC to:

CRITIC($\underline{R}; \underline{w}; \underline{x}; \underline{lambda}$)

To adapt this critic network, we calculate:

CALL ACTION($\underline{R}; \underline{w}'; \underline{x}'; \underline{u}$)
CALL MODEL($\underline{R}, \underline{u}, \underline{noise}; \underline{w}"; \underline{x}"; \underline{R2}$)
CALL CRITIC($\underline{R2}; \underline{w}; \underline{x}; \underline{lambda2}$)
CALL F_MODEL($\underline{lambda2}; \underline{x}''; scratch; F\_R, F\_u, scratch; scratch$)
CALL CRITIC($\underline{R}; \underline{w}; \underline{x}; \underline{lambda}$) $\underline{error} = \underline{lambda2} + \underline{V}(\underline{R2}) - \underline{lambda}$
CALL F_CRITIC($\underline{error}; \underline{x}; F\_w; scratch; scratch$)
$\underline{w} = \underline{w} + learning\_rate * F\_\underline{w}$

where the function $\underline{V}$ is as defined in Section 3.4. To adapt the action network, along the lines of Figure 3.4, we go on to calculate:

CALL F_ACTION($F\_\underline{u}; \underline{x}'; F\_w'; scratch; scratch$)
$\underline{w}' = \underline{w}' + learning\_rate * F\_w'$

3.7.4 Globalized DHP (GDHP)

Section 3.5 proposed that we minimize an error measure based on equation 3.10. For the sake of generality, let us assume that we minimize a weighted sum of square error, where the error for each value of i is weighted by an arbitrary constant A_i. (Those who are disturbed by this may simply use $A_i = 1$, as we did implicitly with DHP.) Our calculations are:

CALL ACTION($\underline{R}; \underline{w}'; \underline{x}'; \underline{u}$)
CALL MODEL($\underline{R}, \underline{u}, \underline{noise}; \underline{w}''; \underline{x}''; \underline{R2}$)
CALL CRITIC($\underline{R2}; \underline{w}; \underline{x}; J2$)
CALL F_CRITIC($1; \underline{x}; scratch; F\_R2; scratch$)
CALL CRITIC($\underline{R}; \underline{w}; \underline{x}; J$)
CALL F_CRITIC($1; \underline{x}; scratch; F\_R; \underline{save}$)
$\underline{error} = \underline{V}(\underline{R2}) + F\_R2 - F\_R$
$delta(i) = error(i) * A(i)$ (for all i)
CALL G_F_CRITIC($\underline{x}; \underline{save}; \underline{delta}; G\_w$)
$w = w + learning\_rate * G\_w$

Adapting the action network is then straightforward. Note how the error vector here is equivalent in meaning to the error vector we used in DHP; the two methods are minimizing the same measure of square error, but in GDHP the lambda vector (i.e., F_R) has to be computed by backpropagation.

References

Barto, A. G., Sutton, R. S. and Anderson, C. W. (1983). Neuronlike elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man and Cybernetics* 13(5):835–846.

Bryson, A. E., Jr. and Ho, Y. C. (1969). *Applied optimal control.* Waltham, MA: Ginn and Co.

Gale, D. (1979). *Theory of Linear Economic Models.* Chicago, IL: University of Chicago Press.

Guez, A. (1987). *Neurocontrollers.* Tutorial C-1 presented at the Third International IEEE Symposium on Intelligent Control, Virginia.

Howard, R. (1960). *Dynamic programming and Markhov processes.* Cambridge, MA: MIT Press.

Klopf, A. H. (1982). *The hedonistic neuron: A theory of memory, learning and intelligence.* Washington, D.C.: Hemisphere.

Lukes, G., Thompson, B., and Werbos, P. (1990). Expectation driven learning with an associative memory. In *Proceedings of the International Joint Conference on Neural Networks.* Hillsdale, New Jersey: Erlbaum, 1:521–524.

Nauta, W., Jr. and Feirtag, M. (1986). *Fundamental neuroanatomy.* New York: W. H. Freeman.

Raiffa, H. (1968). *Decision analysis: Introductory lectures on making choices under uncertainty.* Raeding MA: Addison-Wesley.

Sutton, R.S., (1984). Temporal aspects of credit assignment in reinforcement learning. Ph.D. diss., University of Massachusetts, Amherst.

VonNeumann, J. and Morgenstern, O. (1953). *Theory of games and economic behavior.* Princeton, NJ: Princeton University Press.

Werbos, P. J. (1977). Advanced forecasting methods for global crisis warning and models of intelligence. *General systems yearbook*, Appendix B.

Werbos, P. J. (1979). Changes in global policy analysis procedures suggested by new methods of optimization. *Policy Analysis and Information Systems* 3(1).

Werbos, P. J. (1986). Generalized information requirements of intelligent decision-making systems. In *SUGI-11 Proceedings.* SAS Institute, Cary, NC. (The revised version, available from the author, is easier to read and discusses more issues in psychology.)

Werbos, P. J., (1987a). Learning how the world works. In *Proceedings of the 1987 IEEE International Conference on Systems, Man and Cybernetics*, New York: IEEE Press, I:302–310.

Werbos, P. J. (1987b). Building and understanding adaptive systems: A statistical/numerical approach to factory automation and brain research. *IEEE Transactions on Systems, Man and Cybernetics* 17:7–19.

Werbos, P. J. (1988a). Backpropagation: past and future. In *Proceedings of the IEEE International Conference on Neural Networks.* New York: IEEE Press, I: 343–353. (The transcript of the talk—which is something of a broad, basic introduction—is available from the author.)

Werbos, P. J. (1988b). Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks* I(October):339–356.

Werbos, P. J. (1989a). Maximizing long-term gas industry profits in two minutes in Lotus using neural network methods. *IEEE Transactions on Systems, Man and Cybernetics* 19(2):315–333.

Werbos, P. J. (1989b). Backpropagation and neurocontrol: A review and prospectus. In Proceedings of the IEEE International Joint Conference on Neural Networks. New York: IEEE Press, I:209–216.

Werbos, P. J., (1990a). The consistency of HDP applied to a simple reinforcement learning problem. To appear in *Neural Networks*, March, 1990.

Werbos, P. J., (1990b). Backpropagation through time: What it is and how to do it. To appear in *Proceedings of the IEEE*, August, 1990.

Williams, R. J. (1988). On the use of backpropagation in associative reinforcement learning. In *Proceedings of the IEEE International Conference on Neural Networks*, San Diego, CA.

Wonnacott, T. H. and Wonnacott, R. J. (1977). *Introductory statistics for business and economics.* 2nd ed. New York: Wiley.

Zhu, C., and Skalak, R. (1988). A continuum model of protrusion of pseudopod in leukocytes. *Biophysics Journal* 54(December):1115–1147.