DESIGNING (APPROXIMATE) OPTIMAL CONTROLLERS via DHP ADAPTIVE CRITICS & NEURAL NETWORKS

© George G. Lendaris[1] and Thaddeus T. Shannon[2]
1. Professor, Systems Science, and Electrical & Computer Engineering
2. Graduate Student, Systems Science Ph.D. Program
Portland State University
Portland, OR

1. BACKGROUND

The objective of this chapter is to provide the reader some guidance in applying the Dual Heuristic Programming (DHP) method in the context of designing neural-network controllers. DHP is a member of the class of Critic methods, which in turn is a member of the class of Reinforcement Learning methods. Development of the DHP method benefited from the confluence of several other developments; the following subsections describe associated background ideas useful in appreciating the DHP method. Subsequent sections will describe the DHP method itself, provide suggestions for application of DHP, and present worked-out examples.

1.1 Learning Algorithms

A key distinguishing feature of the computational paradigm known as *neural networks* is its attribute of attaining knowledge via interaction with its environment (vs. having knowledge programmed in). A significant area of research in the neural network (NN) field has been, and continues to be, that of developing strategies by which the NN accomplishes this extraction of knowledge from its environment -- these are typically referred to as *learning algorithms*. These algorithms fall into three general categories (in the following descriptions, the terms 'pupil' and 'teacher' designate, respectively, the NN that is learning, and the process used to accomplish the learning; further, except where indicated, the pupil NN is considered an input/output devise):

1. **Supervised Learning:** This category entails the teacher role having at its disposal full knowledge of the problem context (about which the pupil NN is to learn), and in particular, has available a collection of data pairs (comprising input and associated desired output) with which to conduct the pupil's learning process.
2. **Unsupervised Learning:** In this category, the context is one wherein the notion of input-output does not apply; rather, there is simply a batch of data available from the environment, and it is desired that the pupil NN be assisted (by an implicit teacher role) to discover attributes of the data, which subsequently can be used for a variety of purposes.
3. **Reinforcement Learning:** Whereas in category 1) the teacher role has full knowledge of the output values 'desired' from the pupil NN in response to each given input, here in 3) the teacher does not know the detailed actions the NN should be performing, and can only provide qualitative, sometimes infrequent, feedback about the NN's performance. For example, when we were infants learning to walk, there was no teacher to provide data to us regarding how each of our muscles should have been functioning to accomplish each small movement; rather, only after we took a step and fell down were we provided the general assessment that we 'fell down' (plus some associated information about the form of our falling down). As our walking improved, the kind of feedback (e.g., "wobbliness", "clumsiness", etc.) changed. In the end, based on these 'reinforcement signals,' we learned how to walk. The study of reinforcement learning goes back at least to the early 1960s [Sutton, 1984]. In the context of the present chapter, the important aspect of this category is that the teacher role is endowed with

the capability of providing general measures of performance to the pupil NN (however, hybrid versions allow the teacher to have/provide additional information).

This chapter focuses on application of the Adaptive Critic methodology. The *critic* methodologies are Reinforcement Learning methods that use computational entities to critique the actions of other such entities (the term ‘critic’ was applied in the machine learning context back in 1973 [Widrow, et al, 1973]). In the critic method(s), the teacher role makes substantive use of computations made by a Critic entity. More about this later.

The ‘critiquing’ of the critic methodology normally takes place over time, and a problem context which turns out being a natural for application of the Adaptive Critic method is that of controls, wherein it is desired to design a controller for a ‘plant’ based on specified design criteria, typically involving performance over time. The critic/reinforcement method paved the way for evolving *learning* approaches to solving such controller design problems in the context where little or no *a priori* knowledge of needed controller actions is available, and/or a context wherein substantive changes occur in the plant and/or environment that need to be accommodated by the (learning) controller. In the early stages of this evolution, it was still required to endow the teacher role with some knowledge of the control actions needed (e.g., see [Widrow, et al, 1973]). Continuing up into the early 1980’s, AI researchers who utilized critics were also typically obliged to provide the critic with domain-specific knowledge [Cohen & Feigenbaum, 1982]. In 1983, however, a very important extension was made in which the critic *also* learns, in this case, to provide increasingly better evaluation feedback; this removed the requirement for providing the teacher *a priori* knowledge about desired controller actions [Barto, et al, 1983]. Historically, had the simple term ‘critic’ been used by Widrow and others, the adjective ‘adaptive’ could naturally have been applied to create the term ‘*adaptive* critic’ after the learning capability was added to the critic by Barto, et al. However, Widrow had already used the term Adaptive Critic to imply “learning with a critic”. Nevertheless, the present authors prefer to use the term ‘adaptive’ to refer to the critic’s learning attribute. We note that Werbos [Werbos, 1990b] also uses the term ‘adaptive’ in this latter sense. Barto, et al, cited the checkers-playing program written by Samuel [Samuel, 1959] as an important precursor to their development; they characterized Samuels program as implementing a method “...by which the system improves its own internal critic by a learning process.” The result here was a system in which *two* learning processes are taking place: one for the critic and one for the controller NN. The Barto, et al, context was one of designing a controller with the (not necessarily modest) objective of providing stable control. Now in the 1990’s, a more aggressive objective of using the Adaptive Critic methodology to accomplish design of *optimal* controllers (with stability and robustness included!) is being pursued.

Application of the Adaptive Critic methodology to the design of controllers benefits from the confluence of three developments: 1) Reinforcement Learning, mentioned above, 2) Dynamic Programming, and 3) Backpropagation. Items 2) and 3) are discussed in Sections 1.3 & 1.4.

1.2 Optimal Control

The quest to design controllers that are *best* (in some sense) has been with us for some time. When attempting to make something best (“optimize”), a fundamental concept is that of a criterion function -- a statement of the criteria by which ‘best’ or ‘optimum’ is to be determined. Criterion functions go by various names, ‘cost function’ and ‘utility function’ being typical. These play an important role in later sections.

Entire books have been written to describe various approaches, results, and methods for designing optimal controllers (e.g.,[Athens & Falb, 1966][Bryson & Ho, 1969]). Applications range from designing controllers for linear systems based on quadratic criterion functions, to the more complex non-linear systems. For the linear-quadratic case, complete solutions are available (Ricatti Equation, etc.) and are computationally tractable; for the general non-linear case, while the method known as Dynamic Programming [Bellman, 1957][Howard, 1960][Bertsekas, 1987] is a unified mathematical formalism for developing optimal controls, historically, its application has not been computationally tractable. **The good news** here is that with Adaptive Critic methods, a good approximation to the Dynamic Programming method can be implemented for designing a controller in a system with full non-linear plant capability -- and in a computationally tractable manner.

1.3 Dynamic Programming

Dynamic Programming remains the *only* general approach for sequential optimization applicable under very broad conditions, including uncertainty [Bertsekas, 1987]. While the details of carrying out the Dynamic Programming method are complex, the key ideas underlying it are straight forward. The method rests on Bellman's Principle of Optimality. This Principle stipulates that when developing a sequence of control actions (a "control policy") to accomplish a certain trajectory, say from *a* to *z*, an *optimal* trajectory has the property that no matter how an intermediate point, say *m*, is reached, the remaining trajectory from *m* to *z* must coincide with an optimal trajectory from *m* to *z* as computed from point *m* as a starting point. This turns out being a powerful principle to apply in a critical part of the reasoning path toward attaining the optimal solution.

A necessary early step in designing a controller is to formulate a (primary) utility function $U(t)$ that embodies the objectives for the controlled system in a specified problem context. This step must be done with care -- both from the pragmatics of its computational implementation, but more significant here, from the point of view that the resulting controller's attributes and quality of performance are intimately influenced by this utility function; indeed, the performance of the controlled system is (typically) measured in terms of this utility function.

Application of the Dynamic Programming method begins with defining a *secondary* utility function $J(t)$ in terms of the above (primary) utility function $U(t)$,

$$J(t) = \sum_{k=0} \gamma^k U(t+k) \qquad (1)$$

and the idea is to maximize $J(t)$ (or minimize, depending on whether it represents a "profit" or "cost"), yielding a value designated as J^*. This is known as the Bellman equation. The term γ^k is a discount factor ($0 \le \gamma \le 1$) which allows the designer to weight the relative importance of present vs. future utilities. As will be explained later, for the DHP method recommended herein, γ may be given the value of 1 -- this value is typically not acceptable for other associated methods. A useful identity easily derived from Equation (1):

$$J(t) = U(t) + \gamma J(t+1) \qquad (2)$$

We leave this subsection with the reminder that the Dynamic Programming process is intended to come up with a sequence of controls that is optimal in a stipulated sense, yielding what is called an ‘optimal control policy’, i.e., it is a process of **designing** an optimal controller.

1.4 Backpropagation

A key component of the Adaptive Critic method(s) is an algorithm known as Backpropagation. This algorithm is no doubt the most widely used algorithm in the context of Supervised Learning by feed-forward neural networks. Backpropagation by itself, however, is NOT a training algorithm. Rather, it is an efficient and exact method for *calculating derivatives*. This attribute of Backpropagation is indeed used as the backbone of an algorithm for supervised learning in the NN context. For the purposes of this paper, however, more important is the general view of what Backpropagation is, namely, it is (ingeniously) an *implementation of the chain-rule of taking derivatives*. The order in which associated operations are performed is important, and this prompted its inventor [Werbos, 1974] to use the term ‘ordered derivatives’ for this context.

We will return to this more general aspect of what Backpropagation is a little later, but first, to help understand why taking derivatives is so important in the neural-network learning context, we will go through some background conceptual developments. We consider again the NN as an input-output devise, and create some criterion function (*CF*) to assess the quality of the NN’s output(s) in the given problem context. For the context of learning a specified I/O mapping, the *CF* often used is based on the square of the error between the desired output for the current input and the NN’s actual output for that input. Assume the NN has a feedforward structure (cf. Figure 1), comprising elements with a differentiable, non-decreasing activation function, and each

----

INSERT FIGURE 1 ABOUT HERE

----

layer is fully connected to the next with adjustable weights, depicted by the matrices $\boldsymbol{W}^1$ and $\boldsymbol{W}^2$ in Figure 1. The learning process consists of adjusting the weights until the desired mapping is attained (assuming the desired mapping is possible with the assumed NN structure and activation functions). To be useful, a learning algorithm (weight-adjusting rule) must have associated with it a guarantee that its process will converge to a solution, if one exists. During the 1950s and 1960s, the only learning algorithms with associated theorems of convergence were confined to NNs with a feedforward structure comprising a *single* adjustable layer of weights (Perceptron [Rosenblatt, 1962] and Adaline [Widrow & Hoff, 1960]). Over a decade passed before a major conceptual breakthrough occurred that provided a learning algorithm with proof of convergence for more than a single adjustable layer of weights. As mentioned above, this (supervised) learning algorithm makes use of Backpropagation (more accurately in this context: back-propagation-of-error), and because of this, is often called the Backpropagation *training algorithm* -- it is useful, however, to maintain the distinction being made here. (The Backpropagation algorithm is capable of being applied to recurrent NNs via a method knows as Backpropagation-through-time [Werbos, 1990a]. However this extension is not discussed in the present chapter.)

The essence of figuring out how to adjust the weights in a principled manner to create a learning algorithm is to determine the effect a small change in an individual weight will have on the value of the *CF* -- i.e., some form of $(\Delta CF)/(\Delta w_{ij})$ -- because we want to make a sequence of changes in the weights that will eventually yield some minimum value for the *CF* (or maximum, depending on how it is defined).

A conceptual aid for the mathematics to be developed is as follows: For a neural network with N adjustable weights, construct an N-dimensional vector space, with each dimension corresponding to one of the weights. A *point* in such a space corresponds to a particular setting for all of the weights in the neural network. A process of incremental adjustments to the weights in the neural network will yield a trajectory of points in this N-dimensional vector space. Consider again the *CF*: it will have a particular value that is determined by the specific weights instantiated (and of course, on the mapping that is being learned). As a visualization aid, we add an N+1st dimension to our vector space, and have this new axis represent the value of the *CF*. In principle, we could determine the value of *CF* for each instantiation of weights and plot this value in this N+1 dimensional space (called “Weight Space”), creating a surface over the remaining N-dimensional part of the Weight Space. If N=2 (i.e., just 2 weights in the neural network), it is easy to visualize this surface in a 3-dimensional space, where the x-y plane represents all possible values for the 2 weights, and the surface is plotted in the z direction, representing the values of the *CF* for each pair of weight values. We visualize this surface as having hills and valleys, and our goal would be to find the weights that correspond to the (global) minimum value of the *CF*. If there happened to be only one valley, it is easy to visualize that all we need do from any given starting point of weight values, is to determine the “slope” of the surface over that point, and change the underlying weights such that we have the effect of traveling “down” the surface. This would be done incrementally until the bottom of the valley is reached.

Once we conceptualize the learning problem in this way, then the steps we need to take are clear: 1) develop a method for calculating the *derivatives* (slopes) of the *CF* in the weight space, and 2) make incremental changes to the weights based on this data. A *learning algorithm* (also called *training* algorithm) is crafted to make the incremental changes to the weights designated in step 2. A variety of gradient descent methods have been enlisted to apply the derivative data of step 1 for effecting a learning/training algorithm. While, retrospectively, this conceptualization applies to the Widrow-Hoff “Delta Rule” for adjusting weights in the *single layer* ADALINE that was developed in the 1960s, it took some 20 years for this conceptualization to be clarified sufficiently to be applied to the *multi-layer* neural networks, yielding what was then called the *Generalized* Delta Rule [Rummelhart & McClelland, 1986]. After the Generalized Delta Rule was popularized, it was discovered that the underlying process of taking (ordered) derivatives had previously been developed in a statistics-related Ph.D. dissertation at Harvard University by Paul Werbos in the mid 1970s [Werbos, 1974]. Werbos used the name Backpropagation for the process, so this is now the accepted name in our field for the learning algorithm(s) based on the Backpropagation method of taking derivatives.

As the mathematical derivation of Backpropagation and the associated weight-change algorithm appears in many current neural network textbooks (e.g., see [Rummelhart & McClelland, 1986; Haykin, 1994], it will only be sketched here. The method will then be *described* by use of the *dual* network.

Consider a neural element as shown in Figure 2. Traversing the figure from bottom-to-top, we have the input signals to this *j*th element, which typically come from the outputs of previous

----

INSERT FIGURE 2 ABOUT HERE

----

elements, but for which we use the generic symbol *x* here, $\boldsymbol{x} = \begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix}^T$, the weights into the *j*th neural element $\boldsymbol{w}_j = \begin{bmatrix} w_{j1} & w_{j2} & \dots & w_{jn} \end{bmatrix}^T$, the operation at the "input half" of the *j*th neural element $net_j = \sum_{i=1}^{n} w_{ji} x_i$ (or with vector notation, $net_j = \boldsymbol{w}_j^T \boldsymbol{x}$), and the operation of the "output half" $f_j(net_j)$, where the function $f(\circ)$ is continuously differentiable and non-decreasing, and finally the output

$$o_j = f_j(net_j) = f_j\left(\sum_{i=1}^{n} w_{ji} x_i\right). \tag{3}$$

Let us define a *CF* as a function of o_j, i.e., $CF(o_j)$. As mentioned earlier, a key step in developing a learning algorithm is to determine the effect a small change in an individual weight will have on the value of the *CF*, that is, finding the derivative of *CF* with respect to the w_{ji}. By virtue of Equation (3), we see that the dependence of *CF* on w_{ji} is indirect, so the chain rule of derivatives could be profitably invoked. i.e.,

$$\frac{\partial}{\partial w_{ji}} CF(o_j) = \frac{\partial}{\partial o_j} CF \frac{\partial}{\partial net_j} o_j \frac{\partial}{\partial w_{ji}} net_j \tag{4}$$

We observe that for a given w_{ji} the right-most term evaluates to x_i, the middle term may be represented as $f_j'(net_j)$ (recall our stipulation that $f(\circ)$ be differentiable), and the first term will depend on the actual definition of $CF(o_j)$.

A typical *CF* used is:

$$CF(o_j) = \frac{1}{2}(o_j^d - o_j)^2, \tag{5}$$

where o_j^d is the 'desired' value of the output for the current input. For this case,

$$\frac{\partial}{\partial o_j} CF = -(o_j^d - o_j) \equiv -e_j. \tag{6}$$

Thus, we can write Equation (4) as follows, where we have moved the minus sign to the left side, since we typically want to go "down hill":

$$-\frac{\partial}{\partial w_{ji}} CF(o_j) = e_j \cdot f_j'(net_j) \cdot x_i . \qquad (7)$$

We (arbitrarily) define

$$\delta_j = e_j \cdot f_j'(net_j) , \qquad (8)$$

and we write

$$-\frac{\partial}{\partial w_{ji}} CF(o_j) = \delta_j x_i . \qquad (9)$$

We urge the reader to notice in Equation (8) that δ_j contains the *partial derivative* of the *j*th element's output relative to its net excitation, net_j. Without derivation herein, we will assert later that for elements not in the output layer of the NN, the term corresponding to e_j in Equation (8) will simply comprise the "net excitation" coming into what will be called a *dual* element (to be defined later), whose activation function is linear, given by this *partial derivative*, and hence whose output will be δ_j.

Since we need the derivative of $f(\circ)$, it behooves us to choose a (non-decreasing) function that has an easy derivative to calculate. This has motivated the choice of a sigmoid for the activation function, because its derivative is simply $o_j(1 - o_j)$. Based on the above, a simple weight-change rule ("Delta Rule") would be:

$$\Delta w_{ji} = \beta \delta_j x_i \qquad (10)$$

where β is called a "learning rate", typically less than 1.0, and determines the size of the step to be taken in the direction of the w_{ji} axis in the weight space. By virtue of how we obtained Equation (10), we know that this adjustment in w_{ji} will take us "down hill" in the w_{ji} direction on the *CF* surface.

We comment here that a number of different weight-change rules have been developed based on the above slope data; Equation (10) is just the "vanilla" variety of such rules.

What is known as the Backpropagation training algorithm is based on the above equations. We describe below the algorithm in terms of the *dual* of the given feed-forward NN. (This diagrammatic method of explaining the Backpropagation algorithm is motivated by a similar construct used in [Almeida, 1987] to prove the convergence property of the "backward error propagation technique" when applied to recurrent networks. Almeida, used the term "transposed, linearized perceptron network".)

Generically, we can think of a weight as shown in Figure 3, wherein we show the weight w_{ji}

INSERT FIGURE 3 ABOUT HERE

----

being connected from a source *i* to a destination *j*, with a the signal at its source side x_i, and a *proxy* signal value at its destination side, designated δ_j, that represents some attribute of the destination-neural-element's activity, based on the problem context. The weight-adjustment (or "Delta") rule is simply a product of these two signal values, scaled by a learning rate. In the contexts where Equation (8) is appropriate, we see that $\delta_j = e_j \cdot f_j'(net_j)$, which carries information about a notion of "error" of the *j*th element's output as well as the element's "sensitivity" (slope of its activation function) at the given operating point, net_j.

With this image in mind, all we need for calculating a weight-adjustment increment ("Delta") for each weight w_{ji} is a value for the signal at the weight's input side (x_i) and a value for the *proxy* signal at its output side (δ_j). The **Backpropagation method may be described** as making a "forward" pass through the NN being trained to determine the output values of each neural element (which are "input signals" to the weights connecting that element to the next element along the path) and during this pass, calculating and saving the value $f_j'(net_j) = o_j(1 - o_j)$ for each element (for case of sigmoid activation function). THEN, a pass is made through the *dual* network (described below) to determine the *proxy* signal values, δ_j, for each weight, based on the values of $f_j'(net_j)$ calculated during the forward pass. The (derivative) data gathered during this Backpropagation process is used to calculate weight changes, by application of Equation (10) or some equivalent.

1.4.1 The dual network.

We construct a special NN (called a dual NN) whose purpose is to calculate the δ_j for application of Equation (10). This dual NN has the same *physical* layout as the original NN being trained (see Figure 4), however, the "signals" flow in the opposite ("backward") direction [i.e., the signals come into the top end (corresponding to output side of the original NN) and flow downward

----

INSERT FIGURE 4 ABOUT HERE

(toward what is the input side of the original NN)]. Each neural element has to be "turned around" to accommodate this reversed signal flow, and in addition, the activation function of each element is modified. The activation function is made into a linear one, with slope equal to the value $f_j'(net_j)$ calculated for the corresponding element in the original NN during the forward pass. It is important to note that the weights connecting the elements in the dual NN are the same as the weights in the original NN, albeit the signal flow through them is reversed. All that remains is to apply an "input" at the top end, and make a downward pass through the dual NN. It turns out that the *output* of each *element* in the dual NN provides a value for δ_j corresponding to the *j*th element of the original NN -- without formal derivation, it's useful to remember 3 things here: 1) the activation function of this dual element is linear with slope $f_j'(net_j)$; 2) this dual element will have some net excitation which will be multiplied by $f_j'(net_j)$ to yield the element's output; and 3) we said earlier that δ_j are the product of $f_j'(net_j)$ and some "signal" value.

Thus, by noting the element output values in the original NN generated during the forward pass (the x_i values in the notation of Figure 3), and noting the element output values in the dual NN generated during the "backward" pass (the δ_j values in the notation of Figure 3), we have the numbers needed to apply Equation *(10)* for each of the w_{ji}. [A useful benefit of the dual NN as a means of organizing the needed computations is that the *ordering* necessary for the 'ordered derivatives' being taken is automatically accomplished.]

The remaining item to define here is what "signal" do we apply into the dual NN? The standard Backpropagation training algorithm for feed-forward neural networks was derived using the *CF*:

$$CF = \sum_{j=1}^{\hat{}} \frac{1}{2}\left(o_j^d - o_j\right)^2 \qquad (11)$$

where *k* is the number of elements in the output layer. For *this CF*, using the e_j of Equation (6) for each of the *k* elements in the original NN's output layer, an error vector

$$\boldsymbol{e} = \left[e_1\ e_2\ \dots\ e_k\right]^T \qquad (12)$$

is constructed and used as a "signal" into the dual NN (which has *k* inputs). This input is processed through the dual NN and yields the δ_j values needed for determining the error gradients $\frac{\partial}{\partial w_{ij}}CF$ as mentioned earlier. [Note: since the *error* terms, e_j, are used as signals to feed the dual NN to accomplish the "backward" pass, for *this* learning-algorithm context, Backpropagation is alternatively (and more accurately) called the back-propagation-of-error method.]

1.4.2 Generality of Backpropagation via the dual network.

Generally speaking, a *NN is a function*; indeed, a feed-forward NN whose hidden-layer elements have sigmoid activation functions are known to be 'universal function approximators'. We are sometimes interested in taking derivatives of the function the NN is approximating, for example, with respect to the NN's external inputs. Backpropagation may be used to obtain these, and the calculations can be demonstrated via the dual NN again. In this case, instead of focusing on the δ_j's at the *intermediate* elements of the dual NN for the purpose of calculating the Δw_{ji}, we focus instead on the δ_j's that occur at the dual NN's *output* layer (the bottom layer in Figure 4). Having done this, instead of obtaining error gradients with respect to the w_{ij}, we **obtain error gradients with respect to the original NN's *external* inputs**. This is useful when, say, two NNs (NN-1 & NN-2) are connected in sequence, and NN-1 is to be trained based on error terms available only at the output of NN-2 (cf. Figure 5). In this case, dual NNs for NN-1 and for NN-2 are constructed and connected in (reverse) sequence, and the error terms run ("backwards") through both of them. The element output signals of the NN-2 dual are ignored, and the δ_j's needed for adjusting the w_{ij} in NN-1 are read out of the NN-1 dual.

INSERT FIGURE 5 ABOUT HERE

Also along this line, getting back again to the single NN case, let us use the vector $\boldsymbol{u} = \begin{bmatrix} u_1 & u_2 & \dots & u_k \end{bmatrix}^T$ to represent the outputs of the NN (with *k* elements in the output layer). Then, if we were interested in obtaining a partial derivative of one or more of the outputs u_j (instead of the partial derivative of the *CF* as above), we would simply need to change the input signals to the dual NN. In particular, instead of applying the error vector of Equation (12) to the *k* elements at the input side of the dual NN, we would apply the specialized signal vector $\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \end{bmatrix}_k$ to obtain δ_j values for $\frac{\partial u_1}{\partial w_{ij}}$, the vector $\begin{bmatrix} 0 & 1 & 0 & \dots & 0 \end{bmatrix}_k$ to obtain δ_j values for $\frac{\partial u_2}{\partial w_{ij}}$, …, and finally, the vector $\begin{bmatrix} 0 & 0 & 0 & \dots & 1 \end{bmatrix}_k$ to obtain δ_j values for $\frac{\partial u_k}{\partial w_{ij}}$ (these are easy to verify by starting to take the partial derivatives by hand). Further, as in the previous paragraph, if instead of focusing on the δ_j's at the intermediate elements for the purpose of calculating the Δw_{ji}, we could focus instead on the δ_j's that would occur at the dual NN's output layer (the bottom layer in Figure 4). In this case, the dual NN's overall function (with the specialized input vectors just described) **implements the operator** $\frac{\partial \boldsymbol{u}}{\partial \boldsymbol{r}}$, where $\boldsymbol{u}$ is the vector of outputs of the original NN, and $\boldsymbol{r}$ is the vector of inputs to that NN.

We repeat the emphasis stated at the beginning of Section 1.4 that it has become useful to think of Backpropagation in the more general way(s) described above, namely as a <u>procedure for implementing partial derivatives</u> (explicitly in terms of the chain rule). In this vein, Werbos uses the mathematical-operator notation F_*z* to represent the procedures we described above via the dual NN; specifically, F_*z* = $\frac{\partial^{\circ}}{\partial z} target$, where *target* is known by context (e.g., the *CF* or u_j's of the previous paragraph), the superscript symbol on the right-hand side indicates *ordered* derivatives, and the *z* represents the variables with respect to which the partial derivatives are to be taken (e.g., the w_{ij} or r_j above) [Werbos, McEvoy & Su, 1992][Werbos, 1992]. From the computer programming point of view, Werbos also introduces the vocabulary: *dual subroutine*.

1.5 Heuristic Dynamic Programming.

In his paper entitled "A Menu of Designs for Reinforcement Learning Over Time" [Werbos, 1990b], Werbos observes that, in the context of optimization, "...the biggest challenge is to account for the link between *present* actions and *future* consequences," and commented that there are two basic ways to meet this challenge -- the first being Backpropagation through time (BTT), and the second being 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.” He focused on the Adaptive Critic methods, as we do in this chapter.

For the present context, the key observation (or assertion) made by Werbos was that “...the adaptive critic can be derived as a way of *approximating* Dynamic Programming.” The theory and justification he presented there are not replicated here. In his development, however, he stated that “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.” He observes that *J** is typically too computationally expensive to obtain, and recommends that our only choice is to approximate Dynamic Programming by using a model or network to estimate the *J** function or its derivatives. He goes on to present a sequence of three different “designs” for such Adaptive Critics -- subsequently called Adaptive Critic Designs, or ACDs (see also [Prokhorov, 1997][Prokhorov & Wunsch, 1997][Prokhorov, Santiago & Wunsch, 1995]). From the simplest to the most complex, these ACDs have the following names: 1) Heuristic Dynamic Programming (HDP), 2) Dual Heuristic Programming (DHP), and 3) Global DHP (GDHP). Extensions to each of these ACDs was subsequently made, adding the term ‘Action Dependent’ to each of the above acronyms (e.g., see [Prokhorov & Wunsch, 1997]).

From an implementation point of view, the different ACDs can be distinguished as follows: 1) in the HDP method, the output of the Critic NN is an estimate of the *J* function; 2) in the DHP method, the output of the Critic NN is an estimate of the *derivative* of the *J* function; and 3) in the GDHP method, the Critic NN provides estimates of *both* the function *J* and its derivative as outputs. To distinguish the Action Dependent versions, we note first that only the plant states are used as inputs to the Critic NN for HDP, DHP, and GDHP. In the Action Dependent versions, outputs of the controller (the “action” devise) are also used as inputs to the Critic NN -- hence *action* dependent.

2. THE DUAL HEURISTIC PROGRAMMING (DHP) METHOD

2.1 Preliminaries

After surveying literature on the Critic methods (especially those with comparative experiments), and based on analysis of Equation (18) below, the present authors selected the DHP method as the method of choice for reinforcement learning in the controller-design context. The comparisons made in [Prokhorov, Santiago & Wunsch, 1995] and in [Visnevski & Prokhorov, 1996] both demonstrated superior performance for the DHP over the HDP, with no observable improved performance by the GDHP. The experiments in [Prokhorov, Santiago & Wunsch, 1995] related to an airplane autolander problem context, and the experiments in [Visnevski & Prokhorov, 1996] related to one of the Narendra benchmark problems [Narendra & Mukhopadhyay, 1994]. Private communications from Werbos and McEvoy (cf. [Werbos, McEvoy & Su, 1992]) reported that the DHP method works as well as, or better than, the GDHP on problems of the complexity experimented with so far. They suggested that substantially more difficult problem venues would be needed to “test” the capability of the GDHP method, concluding that it wouldn’t be profitable to use a “sledgehammer to drive small to medium size nails.” We concur. We choose the DHP method.

Further support for the authors' choice of DHP over HDP comes from reviewing Equations (18) and (19). When we get to them later, it will be noticed that these equations make use of a substantial amount of "information" about the system being controlled (via the various different partial derivatives required), and it turns out that this is more information about the system than is *used* in the equivalent equations for the HDP method. With this additional usage of information about the system being controlled, with no apparent equivalent factors on the downside, in our judgement, it makes sense to "place our bets" with the DHP method over the HDP method.

An implementation factor in favor of the DHP method is as follows. As mentioned in the previous section, the HDP critic provides an estimate of *J(t)* at its output, whereas the DHP critic provides an estimate of the *derivatives* of the *J* function at its outputs. Thus for HDP, Equation (2) is used as is, whereas for DHP the derivatives of this equation are used. In considering the recursive nature of this equation and the relationship between *U* and *J*, it becomes clear that the γ term is important, and typically must have a value less than 1 so the *J* term doesn't "swamp" the *U* term during the recursion. This applies in the HDP case. For derivatives of this equation, however, brief consideration allows us to conclude that the derivatives of *U* and of *J* will typically be commensurate (of course, there will always be exceptions), and therefore significance of the γ term reduces substantially. Thus, in the DHP context, the researcher could get away with setting $\gamma = 1$, hence one less parameter to be concerned with in using the DHP method vs. the HDP method. The authors use a value of $\gamma = 1$ in all the experiments reported herein.

An early decision a control engineer must make is whether or not to use a reinforcement-learning method for designing the controller. The decision should primarily be motivated by the kind and quality of information available with which to effect a design, and of course on the kind of control problem at hand. If the information available suffices for use of simpler methods, then it is typically better to use the simpler methods. Once the control engineer decides that a reinforcement-learning method is appropriate, then the suggestion here is for the DHP method to be considered the "workhorse", or default, Adaptive Critic method for general control contexts. One would move "up" to the GDHP, or "down" to the HDP methods only based on analysis and understanding of the problem domain which would provide hints that one of the alternative methods is better to explore. We have no suggestions here concerning the Action Dependent versions of the three methods. Prokhorov [Prokhorov, 1997] suggests that the Action Dependent versions could be useful in the context of recurrent NNs. Recurrent NNs are not dealt with explicitly in this chapter (cf. [Feldkamp, et al, 1997]).

.

2.2 Overview of DHP in Control System Context

The starting point in the control-system context is the 'plant' to be controlled. Clearly, it is useful to obtain as much knowledge about the plant as possible. For the DHP method, a differentiable model of the plant is needed. This can take the form of a set of equations, or if these aren't available, then an identification process is carried out to develop a NN model of the plant.

Next, a clear formulation of the objectives of the controlled system must be made. Then, a utility function U($\boldsymbol{R}$(t),$\boldsymbol{u}$(t)) that embodies these objectives is created ($\boldsymbol{R}$ comprises the state variables, and $\boldsymbol{u}$ the control variables). As mentioned earlier, care must be taken in developing this utility function. A useful hint here is to keep the utility function as simple as possible, consistent with capturing the essence of the control objectives. Avoid the temptation to include more variables in U than is minimally necessary. The more complex the utility function is, the "harder" the critic will have to work to "figure things out". An example of this issue will be given in both of the demonstration problems to be presented in subsequent sections.

The equations and techniques described in this paper are based on a discrete time system or with a discrete sampling of a continuous plant; the usual method of discretizing continuous models of plants is used. For DHP, at least two neural nets are needed, the ***action***NN functioning as the controller, and the ***critic***NN used to train the *action*NN. A third NN must be trained to copy the plant if an analytical description (model) of the plant is not available.

A schematic diagram of important components of the DHP method is shown in Figure 6.

INSERT FIGURE 6 ABOUT HERE

R(t) [dimension n] is the state of the plant at time t. The control signal ***u(t)*** [dimension a] is generated in response to the input ***R(t)*** by the *action*NN. The control signal ***u(t)*** is then applied to the plant and the plant evolves to states ***R(t+1)***. The *critic*NN's role is to assist in designing a controller (*action*NN) that is "good" relative to the objective of the control application --a potential example being: "balance the pole upright and save energy by keeping the control vector's amplitude small". In the DHP method, the *critic*NN estimates the gradient of *J(t)* with respect to ***R(t)***; the letter $\underline{\lambda}$ (a vector) is used as a short-hand notation for this gradient, so the output of the *critic*NN is designated $\underline{\lambda}$. Note: the *1/maxr modules scale the n-dimensional state space R^n to $[-1,+1]^n$; more about this later.

2.3 Updating Process

We refer to Figure 7 to describe the computational steps used in the DHP methodology.

INSERT FIGURE 7 ABOUT HERE

First, we mention that all the boxes in this Figure represent ***roles*** to be performed, and these *roles* can be performed by various "actors". For example, the Critic #1 role performs the mapping of ***R(t)*** to $\underline{\lambda}$(***R(t)***), while the Critic #2 role performs the mapping of ***R(t+1)*** to $\underline{\lambda}$(***R(t+1)***). These roles can be performed by the *same* physical or computational unit, or by *different* such units (neural networks in our case). We will see in the sequel that in some cases we have the exact *same* NN perform both roles; but in other cases, *different* NNs perform the two roles. Similarly for the role called 'plant'. This may be performed by the actual physical plant, a set of equations modeling the plant, or a NN that has been trained to emulate the plant (for most of the work presented in this chapter, we use analytical equations and we take the required derivatives for the reinforcement learning process using these equations -- we remind the reader that a viable alternative is to train up a NN to emulate the plant, and then "do the derivatives" via Backpropagating through this NN model). Similar comments may apply to the remaining boxes in the Figure.

Next, we mention that the boxes in Figure 7 with dark shading mean that we have an analytical expression for that item (again: if an analytical representation of the plant is not available, a NN may be trained up to emulate the plant and used in this role). The clear boxes are neural networks (NNs). The medium shaded boxes represent some critical equations that are to be solved in the learning processes. The dotted lines represent calculated values being fed into the respective boxes. The heavier dot-dash lines indicate where the learning/updating processes occur.

Reading Figure 7 from left to right, the current state ***R(t)*** is fed to the *action*NN, which then generates ***u(t)***. The ***u(t)*** are applied to the plant which then generates ***R(t+1)***. ***R(t)*** is also fed to the critic#1 box and to the utility box, which generate, respectively, $\underline{\lambda}$(***R(t)***) and U(***R(t)***). [Note, in the examples used herein, the Utility functions defined make use of ***R(t)*** but not ***u(t)***; if ***u(t)*** is included in the definition of the utility *U*, then we would show a connection from the output of the *action*NN to the utility box.] After the plant generates ***R(t+1)***, this is fed to the critic#2 box, which then yields $\underline{\lambda}$(***R(t+1)***) -- more simply denoted as $\underline{\lambda}(t+1)$. This value is a key component of the calculations in the medium-shaded boxes, which in turn are needed to perform learning/adaptation of the *action*NN and the *critic*NN. The upper medium-shaded box calculates Δw_{ij}, and the lower medium-shaded box calculates $\lambda^{\circ}(t)$, the “desired” or “target” value for $\underline{\lambda}(t)$ to enable supervised-type training of the underlying critic NN.

We now show the basic equations for the two medium-shaded boxes, and in particular, point out the role of $\underline{\lambda}$(t+1) in those computations.

2.3.1 The upper medium-shaded box of Figure 7.

The underpinnings of the DHP method are the equations for training the NNs. Therefore, it is important to understand how the two NNs are updated. The weights in the *action*NN are updated with the objective of maximizing $J(t)$, and in the present work, a *basic* Backpropagation training algorithm is used (no embellishments) to adjust the weights in the action box. Accordingly, the *action*NN’s weight-adjustment increment is calculated as in the process leading to Equation (10), i.e.,

$$\Delta w_{ij}(t) = lcoef \bullet \frac{\partial}{\partial w_{ij}(t)} J(t) \qquad (13)$$

where
$$\frac{\partial}{\partial w_{ij}(t)} J(t) = \sum_{k=1}^{a} \frac{\partial}{\partial u_k(t)} J(t) \bullet \frac{\partial}{\partial w_{ij}(t)} u_k(t)$$

and
$$\frac{\partial}{\partial u_k(t)} J(t) = \frac{\partial}{\partial u_k(t)} U(t) + \frac{\partial}{\partial u_k(t)} J(t+1)$$

and finally,
$$\frac{\partial}{\partial u_k(t)} J(t+1) = \sum_{s=1}^{n} \frac{\partial}{\partial R_s(t+1)} J(t+1) \bullet \frac{\partial}{\partial u_k(t)} R_s(t+1). \qquad (14)$$

We abbreviate:
$$\frac{\partial}{\partial R_s(t+1)} J(t+1) = \lambda_s(t+1) \qquad (15)$$

and note that $\lambda_S(t+1)$ is obtained from the critic, in response to the input ***R(t+1)***.
The derivatives $\frac{\partial}{\partial u_k(t)} R_s(t+1)$ can be calculated from analytical equations of the plant, if they are available, or by Backpropagation through a previously-trained NN model of the plant.

2.3.2 The lower medium-shaded box of Figure 7.

To train the *critic*NN, whose output is $\underline{\lambda}$, a value has to be calculated for the role of "desired output", here called λ° . Since we don't have a direct way of knowing such a "desired" value, we are obliged to do a kind of "trick". We motivate this as follows. Recalling Equations (1) and (15) and making use of Equation (2) we write

$$\lambda_s^\circ(t) = \frac{\partial}{\partial R_s(t)} J(t) = \frac{\partial}{\partial R_s(t)} (U(t) + J(t+1)) \qquad (16)$$

For utility functions of the form

$$U(t) = \sum_i \left(a_i \bullet R_i^{b_i}(t) \right) + \sum_i \left(c_j \bullet u_j^{d_j}(t) \right) \qquad (17)$$

where a_i, b_i, c_j and d_j are constants, this resolves to

$$\begin{aligned} \lambda_s^\circ(t) = \frac{\partial}{\partial R_s(t)} U(t) &+ \sum_{j=1}^{a} \left(\frac{\partial}{\partial u_j(t)} U(t) \bullet \frac{\partial}{\partial R_s(t)} u_j(t) \right) \\ &+ \sum_{k=1}^{n} \left(\frac{\partial}{\partial R_k(t+1)} J(t+1) \bullet \frac{\partial}{\partial R_s(t)} R_k(t+1) \right) \\ &+ \sum_{k=1}^{n} \left\{ \sum_{j=1}^{a} \left(\frac{\partial}{\partial R_k(t+1)} J(t+1) \bullet \frac{\partial}{\partial u_j(t)} R_k(t+1) \bullet \frac{\partial}{\partial R_s(t)} u_j(t) \right) \right\} \end{aligned} \qquad (18)$$

The partial derivative $\frac{\partial}{\partial R_s(t)} u_j(t)$ is calculated by Backpropagation through the *action*NN. Once λ° is obtained, the "error" components for training the *critic* are calculated as follows:

$$e_s = (\lambda_s^\circ(t) - \lambda_s(t))^2 \qquad (19)$$

2.3.3 Time perspective.

A *perspective* issue arises due to the appearance of the term $\frac{\partial}{\partial R_k(t+1)} J(t+1)$ in the equation for $\underline{\lambda}_s^\circ(t)$; i.e., something from the *future* is needed to provide a target value for the present. We could approach this in (at least) two ways: 1) we could shift our "clock" to (t+1) for the plant so we are always comfortably working "historically", or 2) keep our clock in the present, but use a speeded-up plant model (via analytical equations or a NN) to generate a simulated value for

$R_k(t+1)$ before the real plant moves to the next time step, and apply this to the Critic #2 *role,* which will generate an estimate $\underline{\lambda}(t+1)$, also to be used before the real plant moves to the next time step. When working in a context where everything is done by computer simulation, as for all the work reported herein, there is no pragmatic difference in which perspective is adopted, since everything is taken care of via properly implementing the sequence of computations. In a real-world context however, if the real plant has relatively long "time constants", and if a reasonably good model of the plant is available then it could be useful to adopt the 2nd perspective suggested above. This may be conceptualized as performing the "trick" of jumping to the "future" to get $J(t+1)$ and coming back to the present to manufacture a value for $\underline{\lambda}^{\circ}$ to use as a "desired" output for the purposes of (a supervised) training of the underlying critic NN, which outputs $\underline{\lambda}(t)$. **To recapitulate**, an estimate for $\underline{\lambda}(t+1)$ is generated by the Critic #2 role as response to the ***R(t+1)*** we obtain either by use of a speeded-up plant model, or by shifting our plant's clock one step ahead, and then think in terms of working one time step back. This *perspective* issue requires even more careful attention in cases where the utility function *U* contains state variables that are measured at different points in time, e.g., some *x(t), x(t+1),* & *x(t+2).* This is the case in one of the example problems described later.

2.4 Overview of update procedures.

We now discusses procedures to use the neural net update equations given in the previous sections. First, we take a closer look at the convergence process. In the present notation, $\underline{\lambda}(\boldsymbol{R})$ is the mapping performed by the *critic*NN, $\underline{\lambda}^{\circ}$ is the desired output for the *critic*NN ("calculated" by using the *critic*NN's output in response to ***R(t+1)***), and $\underline{\lambda}\hat{}(\boldsymbol{R})$ is the "solution" (that we don't know) of the Bellman equation and is the target for the other two $\underline{\lambda}$'s. $\underline{\lambda}\hat{}(\boldsymbol{R})$ is a function of the state ***R*** and doesn't change for a time-invariant plant. Since we update the *critic*NN, $\underline{\lambda}(\boldsymbol{R})$ and $\underline{\lambda}^{\circ}$ (which is calculated using the updated *critic*NN) change over time; $\underline{\lambda}^{\circ}(\boldsymbol{R})$ is supposed to converge to $\underline{\lambda}\hat{}(\boldsymbol{R})$, and $\underline{\lambda}(\boldsymbol{R})$ is adjusted in order to converge to $\underline{\lambda}^{\circ}(\boldsymbol{R})$. I.e., $\underline{\lambda}(\boldsymbol{R}) \rightarrow \underline{\lambda}^{\circ}(\boldsymbol{R}) \rightarrow \underline{\lambda}\hat{}(\boldsymbol{R})$. One can imagine this as a tracking problem, where $\underline{\lambda}$ tracks $\underline{\lambda}^{\circ}$. It is important to understand that the better the *critic*NN "solves" the Bellman equation (i.e.: $\underline{\lambda}(\boldsymbol{R}) \longrightarrow \underline{\lambda}\hat{}(\boldsymbol{R})$) the better chance the *action*NN will have to approximate an optimal controller (limited by the capabilities of the *action*NN and the learning process).

Equation (18) for $\lambda_s^{\circ}(t)$ is our principal focus here. If we substitute $\lambda_s(t)$ (i.e., without the superscript) in the left side of Equation (18)) the *critic*NN is considered 'converged' when this new equation *holds true* for all s and all subsequent t's (t can be thought of as an index in the sequence of states).

For convenience of discussion, we paraphrase Equation (18) as follows:

$$: \qquad (20)$$

$$\lambda_s(t) = [\sim Utility] + \sum_{j=1}^{a} ([\sim Utility] \bullet [\sim Action]) + \sum_{k=1}^{n} ([\sim Critic(t+1)] \bullet [\sim Plant]) + \sum_{k=1}^{n} \left\{ \sum_{j=1}^{a} [\sim Critic(t+1)] \bullet [\sim Plant] \bullet [\sim Action] \right\}$$

and with further reduction, we have *(21)*

$$\lambda_s(t) = [\sim Utility] + \sum_{j=1}^{a} ([\sim Utility] \bullet [\sim Action]) + \sum_{k=1}^{n} [\sim Critic(t+1)] \left\{ [\sim Plant] + \sum_{j=1}^{a} [\sim Plant] \bullet [\sim Action] \right\}$$

In this paraphrased way, we make more clear the respective roles of the boxes shown in Figure 7. In a neural network implementation, the *[~Action]* terms of Equations (20) and (21) are calculated via the *action*NN, and the *[~Critic(t+1)]* terms are calculated via a *critic*NN. Take note, however, that the specific form taken by Equations (20) and (21) is dependent upon the form of the utility function assumed in Equation (17).

2.4.1 Conceptual framework for update procedures.

Thinking about and describing the various strategies one might use for performing the controller design process, benefits from starting with a conceptual framework such as the following.

The framework is described in terms of four “loops”.

1. The *plant* ***control*** *loop*. This comprises the controller and plant. The controller observes the state of the plant, makes certain computations, and then issues control actions; the plant responds; the controller observes... and the loop continues.
2. The ***controller training*** *loop*. This is the process of interacting with the controller in a supervised learning context, training it to minimize the secondary utility function *J(t)*. While preliminary training of the controller may be done off-line, when done on-line, the training interactions with the controller must be interlaced with the actions required of the controller in its primary responsibility of controlling the plant.
3. The ***critic training*** *loop*. This is the process wherein the *critic* neural network(s) learns to produce better-and-better approximations to the $\underline{\lambda}$ values needed in loop 2.
4. The (optional) ***plant model training*** *loop*. This is the process wherein a neural-network plant model (if one is used) learns to become a better-and-better representation of the real-world plant -- especially to track changes in the plant’s characteristics.

When learning is being done on-line, the *plant control loop* has highest priority, and each of the other loops must be coordinated to perform their actions in an interlaced fashion. Of course, when doing parts of the design off-line, the engineer gets to choose the appropriate order of actions. In the sequel, we do not include consideration of the *plant model training loop*. This would be a supervised learning situation, where the teacher role simply uses as training I/O pairs the actual inputs and outputs of the real-world plant. The main assumption about this loop is the usual one that the real-world plant's characteristics change at a slow pace in comparison with the response times of the other loops.

For the *plant control loop*, we assume the controller structure has benefited from typical considerations made by a control engineer. With a fully-trained NN controller, we assume the *plant control loop* performs according to specifications. It should be pointed out that developing a "fully-trained" NN controller requires a variety of training stimuli, including a full range of initial conditions and perturbations the plant can be subject to, and to satisfy the 'persistent excitation' requirement. In the case of tracking problems, this includes the full variety of reference behavior the controller is to be used for. Incorporating a representative range of such stimuli into a training simulation must be done on a problem-by-problem basis, taking into consideration the specified primary utility function. Frequently it is useful to begin the training process with stimuli that are "easier", and progressively move to more "difficult" cases.

From an implementation point of view, the challenge spawned by the above is to develop one or more strategies for "putting it all together". Primarily, this involves integrating loops 2 & 3 into the basic implementation of loop 1.The details of evaluating the secondary utility function *J(t)* will dictate the actual time sequencing of all the calculations. The more substantive issues relate to the interleaving of controller and critic network updates (referring back to Figure 7, the controller-update occurs in the *upper* loop following Critic #2, and the critic-update occurs in the *lower* loop following Critic #2). The two primary options are 1) run both training loops simultaneously, or 2) run only one training loop at a time and periodically switch between loops. The first option corresponds to the "classical" strategy [see next section]; and the second option corresponds to the "flip/flop" strategy [see next section]. A variety of hybrid options can be formed by running both loops in parallel as in the Classical strategy (option 1), but performing the updates in ***shadow*** NNs and only occasionally transferring the updates to the NNs used in the functioning system.

Two instantiations of the *shadow* NN concept (not by this name) were reported in [Lendaris & Paintz, 1997] with further results given in [Lendaris, Paintz & Shannon, 1997] and [Lendaris & Shannon, 1998]. In those references, the update strategies using the *shadow* NN concept were simply called Strategy 4a and Strategy 4b. They will be described in the next section, but the key *shadow* NN notion of using two distinct neural networks for the critic is the basis of those strategies. The description provided in Figure 7 was based on applying the *shadow* NN concept to the critic adaptation loop -- cf. implementation of the Critic #1 & Critic #2 roles via *distinct* neural networks. Currently, the *shadow* NN concept is also being applied to the controller adaptation loop; preliminary results of this current work suggest the possibility that even better results may be forthcoming.

Example code structures illustrating the simulation sequence for these options are:

Classical
```
for(i=1; i <= epoch length; i++) {
   controller(i); // generate a control vector
   simulate(i); // simulate the behavior of the plant
   critic(i+1); //  generate λ values for controller training
   calculate target λ(i); //calculate target λs for critic training
   update controller(i); // update the controller network
   critic(i); // generate λ values for critic training
   update critic(i); // update the critic network
}
```

Flip-Flop
```
for(i=1; i <= epoch length; i++) {
   controller(i);
   simulate(i);
   critic(i+1);
   calculate target lambda(i);
   critic(i);
   update critic(i);
}
for(i=1; i <= epoch_length)) i++) {
   controller(i);
   simulate(i);
   critic(i+1);
   update controller(i);
}
```

Shadow Critic
```
for(i=1; i <= epoch length; i++) {
   controller(i);
   simulate(i);
   critic(i+1);
   generate target lambda(i);
   update controller(i);
   shadow critic(i);
   update shadow critic(i);
}
copy shadow critic to critic();
```

Shadow Controller
```
for(i=1; i <= epoch length; i++) {
   controller(i);
   simulate(i);
   critic(i+1);
   generate target lambda(i);
   update shadow controller(i);
   critic(i);
   update critic(i);
}
copy shadow controller to controller();
```

Specific algorithms for the first three structures above are given in the next section, based on prior literature.

References for the Flip/Flop strategy are [Prokhorov & Wunsch, 1996] [Prokhorov, Santiago & Wunsch, 1995] [Santiago & Werbos, 1994] [Visnevski & Prokhorov, 1996] [Werbos, 1992] [Biega & Balakrishnan, 1996]. References for the Shadow Critic strategy are [Lendaris & Paintz, 1997] [Lendaris, Paintz & Shannon, 1997] [Lendaris & Shannon, 1998].

2.4.2 Selected train/update strategies.

Detailed descriptions of various strategies for “solving” (iterating) Equation (18) were given in [Lendaris & Paintz, 1997]. Using Figure 7 again, we describe three of them here: **Classical** corresponds to Strategy 1 of the reference; **Flip/Flop** to Strategy 2; and **Shadow Critic** to Strategy 4. Keep in mind that the output of critic#2 role is required for performing the calculations in the medium-shaded boxes, which in turn must be calculated to perform learning updates in the action and critic#1 roles.

Classical. Straight application of the equation.

In Figure 7, this means that after $\underline{\lambda}$(t+1) is calculated, **both** of the paths leaving critic#2 are traversed, so that the action box and the critic#1 box are updated in each iteration. [Note: In this strategy, the two roles labeled critic#1 and critic#2 are always maintained identical -- i.e., could be the filled by the same physical device, just used for two different calculations.]

Flip/Flop. Basic 2-stage process

During stage 1, train *critic*NN, not *action*NN;

In Figure 7, this means that after $\underline{\lambda}$(t+1) is calculated, only the path which adapts critic#1 is traversed (lower loop), not the path which adapts the action box (upper loop). This is repeated for a designated number of iterations, and then changed to stage 2 (from “flip” to “flop”). As in Strategy 1, critic#1 $\equiv$ critic#2.

During stage 2, train *action*NN, not *critic*NN**.**

In Figure 7, this means that after $\underline{\lambda}$(t+1) is calculated, only the path which adapts the action box is traversed (upper loop), not the path which adapts critic#1 (lower loop). This is repeated for a designated number of iterations, and then changed to stage 1 (from “flop” to “flip”).

Shadow Critic. Single-stage process (as in Classical), with the modification that critic#1 and critic#2 roles are filled by **two physically distinct objects**.

a. Use *critic*NN#2 to update both, *critic*NN#1 and *action*NN.

After $\underline{\lambda}$(t+1) is calculated, adapt both the action box and the critic#1 box as in Strategy 1, however, **leave critic#2 unchanged**. Repeat this for a designated number of iterations (the familiar term ‘**epoch**’ is used here for the designated number of iterations), and *at the end of each epoch, upload the weight values from critic#1 into critic#2,* and continue the process, epoch at a time.

b. Use *critic*NN#2 to update *critic*NN#1; use *critic*NN#1 to update *action*NN.

After $\underline{\lambda}$(t+1) is calculated, adapt the critic#1 box as in Strategy 4a (**leaving critic#2 unchanged**), however this time, apply ***R(t+1)*** to critic#1 to yield $\underline{\lambda}(t+1)$ and use this to update actionNN. Repeat this for a designated number of iterations (epoch), and *at the end of each epoch, upload the weight values from critic#1 into critic#2*, and continue the process, epoch at a time.

2.5 Rationale for the Shadow NNs.

As mentioned above, the “strategies” described in this chapter refer to procedures used to iterate Equation 18. In principle, the straight-forward (“classical”) approach should work just fine -- this is the approach characterized in the previous section as 1) run both training loops simultaneously. In practice, when first attempting to get this procedure to work, one might discover great sensitivity of the process to certain “gain” parameters; it can be difficult to find values for which the learning process converges. We can only speculate that difficulties of this type motivated development of the “Flip/Flop” strategy -- characterized in the previous section as 2) run only one training loop at a time and periodically switch between loops. Nevertheless, the present authors believe that this strategy entails longer training times. This belief is based on the observation that available information is being used less efficiently. Information about both the critic and controller (action NN) is available at each iteration; however, since each loop is put on “hold” while the other is learning, this information is not being used to its fullest. Experiments by the present authors substantiate this prediction. Whereas the controller designs yielded by the Classical and Shadow strategies are roughly equivalent, the Flip/Flop strategy takes approximately twice as long to converge. In the off-line context, this is not a major problem. However, in on-line situations, speed of learning convergence generally *is of essence*.

Another train of thought: The critic is attempting to learn what it can about the plant control loop, and it does so in the face of the controller being changed at every step (Classical method). In addition, a trick is being used of having the critic predict its own next state as a basis to determine a target value to use in a supervised training of the critic -- and this is done in the face of the critic being changed at every step (Classical method). The thought emerges that we might be able to assist the critic in its learning task by providing data from a less “volatile” platform. This leads to the notion of *shadow* NNs. This entails maintaining a version that is *not* being updated each iteration for each NN being trained. From the point of view of the critic, this would allow access to less volatile platforms from which to a) learn about the control loop, and b) to obtain target values for its learning cycle. The Shadow Controller and the Shadow Critic, respectively, are one set of implementation of these ideas. Historically, the Shadow *Critic* method received our attention first, and is the method described herein. Description of and experiments with the Shadow *Controller* methodology will be forthcoming.

Generally speaking, we can report that learning based on the Shadow Critic method does not suffer the learning-rate penalty associated with the Flip/Flop strategy (due to the latter’s less efficient use of available information). Both the Classical and Shadow Critic based strategies yield about twice-as-fast convergence of learning as does the Flip/Flop strategy. Regarding the first difficulty mentioned above, for the problem contexts we have worked on, we have figured out how to get learning based on the Classical update strategy to converge. All three strategies seem to yield about the same quality of controller.

2.6 Assurances for convergence of the training process.

As mention in Section 1, to be useful, a learning algorithm must have associated with it a guarantee that its process will converge to a solution, if one exists. What is our situation in the present context? We have a *double* learning process going on. If we allocate the role of “teacher” to the Critic, and the role of “pupil” to the Controller, then we paraphrase the situation as “the teacher is learning as it is teaching the pupil”. Individually considered, the controller training process seems covered by our existing convergence theorems, inasmuch as there is a bona fide Supervised Learning using the Backpropagation training algorithm, for which convergence theorem(s) exist. But what about the ‘bootstrapping’ kind of situation for the Critic learning? Again we seem to be on safe ground, as there are theoretical results that establish the ‘fundamental soundness’ of this iteration process [Bertsekas & Tsitsiklis, 1996]. In a recent Ph.D. Dissertation, Prokhorov [Prokhorov, 1997] states “Invoking known results from the theory of stochastic optimization, we have just demonstrated that convergence of both critic and action training cycles can be rigorously proven.” This statement seems to give us comfort, but he goes on to say (referring to the question raised above about the *double* learning process going on) “However, we have considered convergence of those cycles taken *separately.*” The answer proposed about the double process is that it “...appears to be rigorously justified only for a single update of the action weights with a small enough learning rate.” We point out to the reader that in principle the theorems of convergence for the Backpropagation training algorithm has a similar constraint, namely, there is a proof based on the theory of stochastic iterative algorithms that assures convergence for one-pattern-at-a-time presentation/update cycles, and another method of proof for full batch training. The middle ground is not rigorously supported by theoretical results (developed to date), yet, the method is successfully used all the time in practice.

We are apparently in a similar situation regarding the *double* learning going on in the Adaptive Critic methods. The theory, so far, supports the micro steps we take, but not the larger-scale issues. Yet, there is empirical evidence (e.g., the examples presented later in this chapter) that the method can converge, and indeed, to useful solutions.

2.7 Assurances for the stability of the control loop during training.

For the control engineer, likely more important questions relate to stability issues. In the case where the controller is trained off-line, can we assure ourselves that the resulting controller-plant combination will be stable? In the case where the controller is trained on-line, can we assure ourselves that the controller-plant combination will be stable at *each* step of the training process? And lastly, might there be an instability induced in this context by virtue of the coupled learning dynamics itself?

A separate chapter of a recent book on neurocontrol [Hrycej, 1997] is devoted to “Stability of Neurocontrollers”. Most of the chapter discusses methods for demonstrating whether the trained-up (nonlinear) controller is stable (for different “kinds” of stability). While not given in the context of Adaptive Critic methods, the methods suggested are in principle applicable, so the reader of the present chapter is directed to that material for useful suggestions. A shorter section of that chapter addresses the topic of *designing* a neurocontroller so that it is provably (under certain caveats) stable. Though that section is short, it gives an answer that seems eminently applicable to the present context. Hrycej suggests the possibility of an approach “...(to) combine the training with the stability proof in order to obtain a controller that is provably stable.” The answer he provides is encouraging: “...it is straightforward. It consists simply in extending the cost function by an appropriately weighted stability constraint penalty term...” It is encouraging to us here because the Adaptive Critic methodology is criterion-function based. While the present authors have not explicitly worked with this suggestion, it seems a workable approach to yield a controller design with stability properties assured. Nevertheless, it still doesn’t necessarily answer our second question that wants assurances about a stable controller-plant combination at *each* step of the training process. Even if the criterion (cost) function successfully enforces a “stability constraint penalty term”, it only means that at the *end* of the training process, a controller will have been designed with this constraint. This method could be useful in an off-line mode which would give us a stable controller to then use in an application, but still leaves open the concern about stability at each step in an on-line training mode.

The answer typically suggested (even by the present authors) is to get a preliminary design for the controller in an off-line mode, so it is “near enough”, and then invoke an on-line training to refine the design to match the actual plant, and which will track incremental changes in the plant’s characteristics over time. We call this “Off-line training for on-line adaptation.” A hint of its applicability is demonstrated in the two examples given later in this chapter.

In the last days of preparation of this chapter, a preprint of an article [Cox & Saeks, 1998] on Adaptive Critic Control came our way. In this article, the authors propose an adaptive critic method that *specifically* addresses the issue of assuring stability at *each* step of the training process. Their approach “reduces the complexity of the adaptive critic algorithm...while simultaneously facilitating the development of stability and robustness criteria. The resultant algorithm...does not require an explicit system identification process.” When fully developed, this could be an important milestone in the application of adaptive critic methods to neurocontrol. Their approach is not the same as the DHP method described herein. The present authors plan to explore the details of their proposal with an eye toward incorporating the ideas into the DHP method.

Regarding the third question in the first paragraph of this Section, we are not aware of any rigorous treatment of this issue. However, we are confident that as deeper interaction begins to take place as we couple more and more learning loops, this issue will have to be successfully dealt with.

3.0 GENERAL APPROACH FOR APPLICATIONS

3.1 Specify the desired behavior via the primary utility function.

As discussed in Section 2.2, the first step in designing a controller using the DHP method is to define a (primary) utility function that embodies the objectives of the controlled system. This is an important step. Examples are provided in Sections 4.1 and 4.2. The secondary utility function defined by Equation (1) is then constructed. This becomes the *CF* (criterion function) for training the controller (action NN). When crafting the primary utility function, keep in mind practical issues related to the requirement of DHP that the derivatives of *J(t)* with respect to the plant states are to be approximated. In particular, it will be important that meaningful terms in the expansion

$$\frac{\partial}{\partial R_k(t)}J(t) = \frac{d}{dR_k(t)}U(t) + \frac{d}{dR_k(t)}J(t+1) \qquad (22)$$

be available without too much delay. In particular, the (total) derivative of $U(t)$ with respect to the plant states should be easy to calculate numerically without having to wait an inordinate period of time.

3.2 Define the controller (action NN) architecture.

In general, from a black box point of view, the inputs to the action NN will be the state variables of the plant, and the outputs of the action NN will be the control (manipulated) variables, with the NN performing a mapping between these two sets of variables. If not all the state variables are available, then the inputs to the NN should comprise those state variables that are observable, plus sufficient lagged values of one or more of the observable state variables to provide the needed information. For tracking problems, the desired behavior must also be included in the NN’s inputs.

The number of inputs and the number of outputs of the action NN are thus specified by the problem context. Assuming a feed-forward structure is being used, when selecting the number of hidden layers and their respective sizes, keep in mind that smaller structures are advantageous in that they train faster, are computationally less expensive, and tend to have better generalization performance than more complex architectures do (this latter statement assumes both structures do equally well on the training data). Obviously, the NN's structure must be of sufficient size/complexity to learn the desired mapping. Keep in mind also that one of the structure-related choices available is the inclusion (or not) of trainable bias terms.

It is useful to scale all the input variables into a range appropriate for the NN's processing elements. This was mentioned in Section 2.2 relating to the scaling blocks in Figure 6. If this step is not taken *a priori*, the initial layer of the NN's weights will have to do the scaling, which complicates the initial stages of training. While the scalings can be determined analytically in some cases, frequently it may be necessary to determine them empirically by observing the plant's behavior.

The action NN's output layer may use linear activation functions in its elements, however there is sometimes better training success with sigmoid-type activation functions. In this *latter* case, the outputs are constrained to the range of [-1,1] or [0,1], and accordingly these values may need to also be scaled in order to transform this limited range into the range required of the control variables to be applied to the plant. This would be in addition to the scaling blocks suggested in Figure 6.

Finally, the range of values over which the NN's weights are randomly initialized can affect the learning characteristics. Starting with a very small range around zero is usual, but experience with a particular plant may suggest other initial starting ranges.

3.3 Define the critic NN architecture.

The set of inputs to the critic NN should include just those variables needed by the critic to approximate the secondary utility function. Making this (minimal) selection is often easier said than done. However, a good starting point are the plant's state variables that appear in the primary utility function and the reference variables provided to the controller in the tracking context. Beyond this, engineering judgement (and sometimes "tinkering") is called for.

The outputs of the critic NN represent the (estimated) values of the derivative of *J(t)* with respect to either all or some subset of the plant's states. The number of critic NN outputs, then, is determined by the fuzzy criterion that we need just "enough" information to Backpropagate through the plant (well, really the plant's surrogate: either analytic equations or a NN model, cf. Section3.4) to do a good job of designing the controller. If the primary utility function *U(t)* contains state variables at different time instants [e.g., see Section 4.2], then for the partial derivatives $\frac{\partial}{\partial R_k(t)} J(t)$, the time index for the $R_k(t)$ will have to reflect this.

Comments regarding other design choices are similar to the previous section, namely, a sufficient but not excessive number of hidden layer elements should be included in the critic NN; scaling factors for the critic NN will usually be identical to those used for the same variables in the action NN's inputs; and the initialization range for the critic NN's weights is again relevant, and is determined empirically.

3.4 Implement the differentiable model.

A differentiable model of the plant's behavior is required at two points in the DHP process. We see this clearly in Equation (20), by noting that the symbol *[~Plant]* shows up in two places. Referring back to Equation (18), we see that these are for evaluating the $\frac{\partial}{\partial R_s(t)} R_k(t+1)$ and the $\frac{\partial}{\partial u_j(t)} R_k(t+1)$ terms. The first of these terms refers to all the partials of the plant state variables with respect to each of the other plant state variables; the second term refers to all the partials of the plant state variables with respect to the control variables. All of these partials, for all j,k and s, must be obtained from a (differentiable) plant model. If an analytic description of the plant state exists, the partials can be calculated from these equations; alternatively, Backpropagating through a trained-up NN model of the plant may be done. When using an analytic model, care must be taken to effect the proper "ordering" when evaluating the derivatives. When using an NN model, this "housekeeping" is automatically taken care of in the dual NN (cf. Section 1.4.1).

3.5 Define the simulation sequence.

We visit the first three of the "loops" defined in Section 2.4.1:

1. The *plant control loop* is the basic component of a simulation. At specific points in time, the plant state data is presented to the critic NN, which then generates a control signal. The control signal is then applied to the plant over the following time interval until a new control signal is generated. The practical issues in this loop are the simulation method for evolving the plant states through time, and the duration of the control interval.

2. The *controller training loop* is the process of providing feedback to the controller neural network in a supervised learning context. The controller is trained to minimize the secondary utility function *J(t)*. A simple gradient based training method for the critic NN can be used as soon as the error data are available from the critic in the form $\frac{\partial}{\partial u_k(t)} J(t+1)$. From Equation (14), we see that

$$\frac{\partial}{\partial u_k(t)} J(t+1) = \sum_{s=1}^{n} \lambda_s(t+1) \bullet \frac{\partial}{\partial u_k(t)} R_s(t+1) \qquad (23)$$

So the Action NN can be updated at time *t* by using the output of the critic NN at time *t+1* for generating error terms. Standard additions to the basic gradient descent training method, such as the addition of a momentum term and/or the use of offset factors to keep the derivatives of the activation functions away from zero, may prove helpful. Alternative methods of NN training may also be used, e.g., see [Visnevski & Prokhorov, 1996].

3. The *critic training loop* provides feedback to the critic NN and also updates the critic's internal weights so as to produce better approximations to the $\underline{\lambda}$ values which are to be used in the in the *controller training loop*. The calculation of target, or "desired", values for $\underline{\lambda}$ from which error terms for training the critic NN are developed have been discussed in Section 2.3.2.

Ensuring an adequate variety of training stimuli is important for controller training. The stimuli should cover both the full range of initial conditions and perturbations the plant can be subjected to, and in the case of tracking problems, the full variety of reference behavior the controller is to be used for. Generating a representative set of such stimuli for the training is of course problem specific, and the form of the crafted primary utility function must be taken into account. Frequently, it is useful to begin the training process with stimuli that represent "easier" control situations, and progressively move to more and more "difficult" control situations.

4.0 EXAMPLES

4.1 Pole-Cart benchmark example.

The pole-cart balancing problem is a simple system which demonstrates unstable nonlinear behavior. The standard setup of the system is to place a pole on a cart that rides along a straight section of track (cf. Figure 8) The pole is constrained so that it can only pivot along the axis of the track. The control problem is to determine the sequence of forces that when applied to the cart (horizontally) will keep the pole balanced upright. Additionally, a target position for the cart on the track can be specified, and the total force applied could be required to be minimized.

INSERT FIGURE 8 ABOUT HERE

The state of the system is uniquely specified by the position and velocity of the cart, and by the angular position and velocity of the pole. The second order system equations are then:

$$\ddot{\Theta}(t) = \frac{g\sin\theta_t + \cos\theta_t\left[\frac{-F - ml\dot{\theta}_t^{2}\sin\theta_t + \mu_c sgn(\dot{x}_t)}{m_c + m}\right] - \frac{\mu_p\dot{\theta}_t}{ml}}{l\left[\frac{4}{3} - \frac{m(\cos\theta_t)^2}{m_c + m}\right]} \qquad (24)$$

and

$$\ddot{x}_t(t) = \frac{F_t + ml\left[\dot{\theta}_t^{2}\sin\theta_t - \ddot{\theta}_t\cos\theta_t\right] - \mu_c sgn(\dot{x}_t)}{m_c + m} \qquad (25)$$

where

F is the force applied to the cart (10 x Newtons),

g is the acceleration due to gravity (m/s^2),

m_c is the mass of the cart (kg),
m is the mass of the pole (kg),
l is the length of the pole (m),
μ_c is the coefficient of friction of the cart on the track,
μ_p is the coefficient of friction of the pole on the cart.

The base system parameters used in the rest of this example are:
$g = 9.8m/s^2$,
$m_c = 1.0kg$,
$m = 0.1kg$,
$l = 0.5m$,
$\mu_c= 0.0005$,
$\mu_p=0.000002$.

The desired state, the pole balanced upright and with the cart stationary at the desired location on the track, is an unstable fixed point of the system. In the control context, the desired system state can be expressed with a utility function of the form

$$U(t) = -\alpha[\theta(t) - \tilde{\theta}(t)]^2 - \beta[x(t) - \tilde{x}(t)]^2 \qquad (26)$$

where α and β are arbitrary weighting constants, $\tilde{\theta}(t)$ and $\tilde{x}(t)$ are the desired values for pole angle and track location for the cart. The constants allow the relative importance of track position to pole position to be varied. Additional terms can of course be added to this function to represent criteria such as minimizing total force used, minimizing the maximum acceleration applied to the pole, etc.

The plant is simulated by integrating the plant equations through time starting from a specific set of initial conditions. Given initial position and velocity values, integration produces the plant's trajectory through time. A variety of integration techniques can be used, the choice of techniques involves picking a particular trade off between accuracy and computational speed. A simple first order technique like Euler integration can be used here with little loss of generality, though slightly more complicated Runga-Kutta techniques can also be used. Whatever the technique employed, one must also choose the length of the time step to be used in the integration. While smaller time steps produce more accurate results, they also require more calculations to produce a system trajectory for a fixed time interval. The obvious constraint on integration step size is that it must be less than the sampling interval used by the controller.

4.1.1 Defining The Utility Function.

The difficulty of the control task can be varied by including different terms of the state variables in the utility function, and by adjusting the length of track available for travel. The simplest utility function would comprise just the first term of Equation (26). Our *desired angle* is 0, and we arbitrarily choose $\alpha = 0.25$, yielding

$$U(t) = -0.25\theta^2(t) \tag{27}$$

In this case, there are no limits placed on the length of track available for the cart to travel over. We call this the θ-only problem since only the θ(t) term appears in the utility function, and no limits are placed on the position x(t) of the cart. There are many obvious ways to introduce position states into the utility function; they all make the problem more difficult for the controller, yet also more realistic. The simplest is to adopt a utility function like

$$U(t)) = -0.25\theta^2(t) - 0.25x^2(t) \tag{28}$$

i.e., $\tilde{x}(t) = 0$. This θ–x problem is the obvious first extension of the θ-only problem.

4.1.2 Controller Implementation.

In the pole-cart problem we have only one manipulated variable, so a basic network architecture for our controller has four inputs, several hidden layer elements, and a single output element. An alternate controller might also include both acceleration variables in its inputs. An example of both kinds of controllers will be given for the x-θ problem, and their performance compared.

For the θ-only problem, notice that the only linkage of the position variables to the θ-only utility function is through the incredibly small friction coefficient terms in the expression for $\ddot{\theta}$. These terms depend on the sign of $\dot{x}$ and so their partial derivatives with respect to $\dot{x}$ are zero almost everywhere. Thus for this limited problem, the system can be decomposed and we can dispose of the position variables in both the controller and the critic networks. This significantly simplifies controller and critic training. Two controllers for the θ-only problem will be illustrated, one using only angular position and velocity as inputs, the other also including angular acceleration.

Each network input and output requires scaling from the variable's range of measurement into the interval [-1,1]. Scaling factors for the controller inputs were derived by noting the range of state variable values actually observed for the base set of system parameters. The ranges to be scaled are obviously symmetric around zero and are:

x : ±4.0

$\dot{x}$: ±4.0

$\ddot{x}$: ±12.0

θ : ±1.6

$\dot{\theta}$: ±3.5

$\ddot{\theta}$: ±15.0

For the controller output, the scaling factor represents the controller's maximum gain, i.e. the maximum force the controller can exert on the cart. This factor will end up limiting the maximum angle from which the controller can return the pole to a vertical position. A higher gain value should enable the controller to bring the pole upright from a larger initial deflection from vertical, but it will also make it much easier for the controller to overcorrect for a small deflection. For this example we adopt a gain value of 15, i.e. the maximum force the controller can exert on the cart is 15 Newtons. Note that fine controllers can be trained using both higher and lower gain values.

4.1.3 Critic implementation.

For the θ-only problem, the critic network may have either two or three inputs $\theta(t)$, $\dot{\theta}(t)$, and $\ddot{\theta}(t)$, and correspondingly two or three outputs $\lambda_1(t) = \frac{\partial}{\partial\theta(t)}J(t)$, $\lambda_2(t) = \frac{\partial}{\partial\dot{\theta}(t)}J(t)$ and $\lambda_3(t) = \frac{\partial}{\partial\ddot{\theta}(t)}J(t)$. Only one hidden layer processing element is needed. The hidden layer element uses a hyperbolic-tangent activation function with no bias, while the output elements use linear activation functions with no bias. The inputs are scaled using the above factors.

The somewhat more complex θ-x problem uses a four-or-six input, four-or-six output critic network. The variables available for inputs are $x(t)$, $\dot{x}(t)$, $\ddot{x}(t)$, $\theta(t)$, $\dot{\theta}(t)$, and $\ddot{\theta}(t)$; the outputs would be $\lambda_1(t) = \frac{\partial}{\partial x(t)}J(t)$, $\lambda_2(t) = \frac{\partial}{\partial\dot{x}(t)}J(t)$, $\lambda_3(t) = \frac{\partial}{\partial\ddot{x}(t)}J(t)$, $\lambda_4(t) = \frac{\partial}{\partial\theta(t)}J(t)$, $\lambda_5(t) = \frac{\partial}{\partial\dot{\theta}(t)}J(t)$ and $\lambda_6(t) = \frac{\partial}{\partial\ddot{\theta}(t)}J(t)$. As above, the hidden layer has a single element with a hyperbolic-tangent activation function and no bias, while the output elements are linear with no bias. All the inputs are scaled as above.

4.1.4 Model implementation.

Since the system equations are known, we use the following set of derivatives as our model:

$$\frac{\partial}{\partial \boldsymbol{R}(t)}\boldsymbol{R}(t+1) = \begin{bmatrix} \frac{\partial x_{(t+1)}}{\partial x_t} & \frac{\partial x_{(t+1)}}{\partial \dot{x}_t} & \frac{\partial x_{(t+1)}}{\partial \ddot{x}_t} & \frac{\partial x_{(t+1)}}{\partial \theta_t} & \frac{\partial x_{(t+1)}}{\partial \dot{\theta}_t} & \frac{\partial x_{(t+1)}}{\partial \ddot{\theta}_t} \\ \frac{\partial \dot{x}_{(t+1)}}{\partial x_t} & \frac{\partial \dot{x}_{(t+1)}}{\partial \dot{x}_t} & \frac{\partial \dot{x}_{(t+1)}}{\partial \ddot{x}_t} & \frac{\partial \dot{x}_{(t+1)}}{\partial \theta_t} & \frac{\partial \dot{x}_{(t+1)}}{\partial \dot{\theta}_t} & \frac{\partial \dot{x}_{(t+1)}}{\partial \ddot{\theta}_t} \\ \frac{\partial \ddot{x}_{(t+1)}}{\partial x_t} & \frac{\partial \ddot{x}_{(t+1)}}{\partial \dot{x}_t} & \frac{\partial \ddot{x}_{(t+1)}}{\partial \ddot{x}_t} & \frac{\partial \ddot{x}_{(t+1)}}{\partial \theta_t} & \frac{\partial \ddot{x}_{(t+1)}}{\partial \dot{\theta}_t} & \frac{\partial \ddot{x}_{(t+1)}}{\partial \ddot{\theta}_t} \\ \frac{\partial \theta_{(t+1)}}{\partial x_t} & \frac{\partial \theta_{(t+1)}}{\partial \dot{x}_t} & \frac{\partial \theta_{(t+1)}}{\partial \ddot{x}_t} & \frac{\partial \theta_{(t+1)}}{\partial \theta_t} & \frac{\partial \theta_{(t+1)}}{\partial \dot{\theta}_t} & \frac{\partial \theta_{(t+1)}}{\partial \ddot{\theta}_t} \\ \frac{\partial \dot{\theta}_{(t+1)}}{\partial x_t} & \frac{\partial \dot{\theta}_{(t+1)}}{\partial \dot{x}_t} & \frac{\partial \dot{\theta}_{(t+1)}}{\partial \ddot{x}_t} & \frac{\partial \dot{\theta}_{(t+1)}}{\partial \theta_t} & \frac{\partial \dot{\theta}_{(t+1)}}{\partial \dot{\theta}_t} & \frac{\partial \dot{\theta}_{(t+1)}}{\partial \ddot{\theta}_t} \\ \frac{\partial \ddot{\theta}_{(t+1)}}{\partial x_t} & \frac{\partial \ddot{\theta}_{(t+1)}}{\partial \dot{x}_t} & \frac{\partial \ddot{\theta}_{(t+1)}}{\partial \ddot{x}_t} & \frac{\partial \ddot{\theta}_{(t+1)}}{\partial \theta_t} & \frac{\partial \ddot{\theta}_{(t+1)}}{\partial \dot{\theta}_t} & \frac{\partial \ddot{\theta}_{(t+1)}}{\partial \ddot{\theta}_t} \end{bmatrix} \tag{29}$$

$$= \begin{bmatrix} 1 & \tau & 0.5\tau^2 & 0 & 0 & 0 \\ 0 & 1 & \tau & 0 & 0 & 0 \\ 0 & 0 & 0 & \partial\ddot{x}_{\theta,t} & \partial\ddot{x}_{\dot{\theta},t} & \partial\ddot{x}_{\ddot{\theta},t} \\ 0 & 0 & 0 & 1 & \tau & 0.5\tau^2 \\ 0 & 0 & 0 & 0 & 1 & \tau \\ 0 & 0 & 0 & \partial\ddot{\theta}_{\theta,t} & \partial\ddot{\theta}_{\dot{\theta},t} & 0 \end{bmatrix} \tag{30}$$

and

$$\frac{\partial}{\partial u(t)}\boldsymbol{R}(t+1) = \begin{bmatrix} \frac{\partial x_{(t+1)}}{\partial u_t} & \frac{\partial \dot{x}_{(t+1)}}{\partial u_t} & \frac{\partial \ddot{x}_{(t+1)}}{\partial u_t} & \frac{\partial \theta_{(t+1)}}{\partial u_t} & \frac{\partial \dot{\theta}_{(t+1)}}{\partial u_t} & \frac{\partial \ddot{\theta}_{(t+1)}}{\partial u_t} \end{bmatrix}$$

$$= \begin{bmatrix} \alpha_t(0.5)dt^2 & \alpha_t dt & \alpha_t & \beta_t(0.5)dt^2 & \beta_t dt & \beta_t \end{bmatrix}$$

where $\alpha_t = \frac{1}{m_c + m}$, $\beta_t = \frac{-\alpha_t \cos^2\theta_t}{\left[\frac{4l}{3} - ml\alpha_t\cos\theta_t\right]}$, with $\partial\ddot{x}_{j,t}$ and $\partial\ddot{\theta}_{j,t}$ being numerical approximations obtained by forward propagating small perturbations in state variable *j* through the plant simulation.

4.1.5 Putting it together.

1) Given either of our chosen utility functions, there are no overt time delays in the systems. Therefore we can use a direct implementation of any of the four strategies detailed in Section 2.4.

2) We need to fix a controller sampling interval for the simulation. Our base case sampling interval is 0.05 seconds, but we investigate the effect of this parameter both on the DHP training process and on the quality of the trained controller.

3) Both the controller and the critic networks are updated using a basic gradient descent method, with no momentum terms used in either case. A (derivative) offset value of 0.1 is used for training the critic network, but no offset is used when doing the controller weight updates. A variety of controller and critic learning rates were used. The results reported here are based on controller and critic learning rates of 0.5 and 0.1 respectively for the θ-only problem and 0.05 and 0.01 respectively for the θ-x problem.

4) Training stimulus is provided by starting the plant simulation in an initial state [format $(x, \dot{x}, \theta, \dot{\theta})$] of $(0, 0, \varepsilon_1, 0)$ and letting the system run for 30 seconds of simulation time. The plant is then reset to a new initial state $(0, 0, \varepsilon_2, 0)$ and run for 30 more seconds. Whenever the pole falls all the way over (an absorbing state of the system), the plant is reset to the initial state for the current training period, and started over.

For example, the simulation is started with the cart stationary at the zero location on the track with the pole set at an angle of 5 degrees from vertical with no initial velocity. Training commences and if the pole falls over, it and the cart are reset to the initial state, and the training continued. Once 30 seconds has elapsed in the simulation (regardless of the number of "drops"), the cart is recentered on the track, and a new 30 second cycle is begun with the pole set at 2.5 degrees from vertical, in the opposite direction, with no initial velocity.

The sequence of initial angles (ε_j) used for training is 5°, -2.5°, 7.5°, -10°, 2.5°, and -7.5°. This sequence is repeated three times. Training is then halted, and the performance of the controller is measured on each of the training initial conditions. A second set of tests (for generalization) is then performed using a different set of (substantially larger) starting angles: -38°, -33°, 23°, and 38°.

Two observations can be made concerning these training stimuli. First, use of high-magnitude training angles makes the training process more difficult to converge. Second, lack of adequate training stimulus can lead to poor generalization. These observations support the practice of starting the training process off on small angles and then progressively increasing the difficulty of the training tasks.

4) This set of training parameters, network structures and training stimuli has been selected, based on experience, to produce convergence to a satisfactory controller with probability 1. Using higher learning rates, a wider range for weight initialization, larger training angles, etc., often lead to lack of convergence for the learning process -- i.e., results in a non-performing NN. An occasional convergence may occur (since there are random aspects to the learning process, this is possible), and some times the learning occurs rather fast in these cases. While for a practical problem

a single well trained controller network (based on some “lucky” training session that converged) may be all that is desired, but in general, when applying a methodology it is nice to know that there is an implementation scheme available (i.e., combination of values of parameters mentioned above) that will always produce reasonable solutions.

4.1.6 Simulation results.

A snapshot of the initial training sequence for a (3-input, 1-hidden element, 1-output *controller* NN; 3-input, 1-hidden element, 3-output *critic* NN) combined architecture, abbreviated (3-1-1; 3-1-3), for solving the θ-only problem using the Classical strategy with the above parameters is illustrated in Figure 9a. One can see that after an initial series of drops, the controller very quickly learns to not drop the pole. Further refinements of the controller are then achieved over the remainder of the training sequences. The response of the trained controller when presented with a large initial angle in a generalization test is shown in Figure 9b.

INSERT FIGURE 9 ABOUT HERE

A similar result for a (6-1-1, 6-1-6) combined architecture for solving the θ-x problem is shown in Figure 10a. For this controller, a further generalization test is to present an initial condition with no angular deviation but with the cart placed away from the desired track location. Moving the cart to the desired track location while keeping the pole balanced is not a task that this controller was explicitly trained on. Yet, as can be seen in Figure 10b, the controller’s initial response was to move the cart *away* from the desired track position so as to get the pole moving in a way that allows the controller to get the cart to come to rest at the desired track location and the pole to return to vertical at the same time!

INSERT FIGURE 10 ABOUT HERE

This 6-1-1, 6-1-6 architecture can be compared to alternate architectures, shown in Figure 11a to Figure 11f. Shown in order are a 4-1-1, 4-1-4 architecture that includes only position and velocity information, the 6-1-1, 6-1-6 architecture which also includes acceleration information, and a 6-3-1, 6-3-6 architecture with expanded hidden layers. Figure 11a through Figure 11c are step responses of selected training angles, while Figure 11d through Figure 11f are step responses for cart displacement generalization tests. It is clear that while all three controllers can accomplish the control objective in each case, the step response of the 6-1-1 controller (trained by the 6-1-6 critic) is visibly superior to the others.

INSERT FIGURE 11 ABOUT HERE

An overall comparison of training time and generalization ability across a range of architectures is provided in Figure 12a and Figure 12b for the θ-only problem and in Figure 13a through Figure 13c for the θ-x problem. We note that in the θ-only case, the average number of drops during training decreases as the architecture complexity increases, while the generalization performance increases (decreasing cost/errors) somewhat with the addition of acceleration information. For the θ-x problem, however, while the number of drops during training also decreases with an increase in network complexity, there is a dramatic degradation of generalization performance (increased cost/errors) as more hidden layer elements are added to the networks.

INSERT FIGURE 12 &13 ABOUT HERE

Also note that the performance of the Classical, Flip-Flop and Shadow Critic training strategies can be compared in these graphs. These results support the general statement that the Flip-Flop strategy takes twice as long to achieve equivalent results as the Classical or Shadow strategies. A fair comparison of the resulting controller performance, however, should not be made from these graphs, as a fixed amount of training time was allowed in each case. If our above observation is correct, the Flip-Flop strategy should be allocated more training time to refine its controller design before it can be fairly compared with the controllers produced by the other strategies.

The rate at which the controller operates has obvious computational ramifications -- the faster the sampling rate, the higher the computational requirements. On the other hand, it is equally clear that too slow a sampling rate will result in poor control. It is thus of import to explore the effect of different sampling rates on the performance of the DHP process and of the resulting controllers. Sweeps through the controller sampling interval parameter are illustrated in Figure 14 and Figure 15.

INSERT FIGURE 14 &15 ABOUT HERE

As expected, it was found in general that the quality of control continued to improve as the time step size decreased, up to some asymptotic level. Further, it was found that the training performance measures also improved as the time step size decreased, again up to some asymptotic level.

From the perspective of the DHP method, faster sampling rates allow more learning to occur in a fixed time period (up to some limit), since there are more weight updates in that time period. Conceptually, the limiting upper value of the sampling rate is dependent on the maximum rate at which the plant can provide “appropriate information” to the DHP method upon which to design the controller -- i.e., dependent upon the plant’s time constants.

Various experiments were run for both the θ-only and the θ-x cases, using six different sampling intervals, ranging from 0.003125 sec. to 0.1 sec., each separated by a factor of 2. For ease of comparison, the same learning rates were used for all the experiments. In general, however, for the high sampling-rate cases, smaller learning rates produce smoother convergence of the DHP process.

For the θ-only case, the results yielded a rather smooth diminishing-return type of plot, from which one would pick the 0.0125 or 0.00625 sampling period.

For the θ-x case, while the general results are the same, some ambiguity appears in the generalization tests for the coarse sampling time interval of 0.1 sec. In this case, if the system hasn't learned to balance the pole yet, then the position-related performance measurements turn out to be very bad. In any case, for the θ-x case also, one would pick the.0125 (or 0.00625) sampling interval.

4.1.7 On-line adaptation.

An important potential for the Adaptive Critic methods will be their applicability to on-line learning -- that is, to be able to adapt to substantive changes in the attributes of the plant being controlled. Results from both this Pole-Cart benchmark problem and the Narendra benchmark problem described in the next section, provide some very encouraging results. We have not seen experimental results of this nature reported in the literature. What we have, in a sense, is a situation wherein the controller "learns off-line and then adapts on line". In this Pole-Cart context, the experiment we crafted was as follows: for the controller (6-1-1) that was trained-up for a pole length of 1 meter, the length/mass of the pole was increased in varying amounts to determine the robustness of the controller design (i.e., with no on-line learning turned on). It was observed that the controller successfully (i.e., no drops) balanced the pole for increases in the length of the pole from a starting value of 1 meter all the way up to approximately 2 meters. Above 2 meters, the pole would sometimes fall. To test the DHP method's ability to operate on line -- e.g., to effect an adaptive refinement of the controller design -- tests were run wherein the DHP learning process was left on as the test displacements were applied to the pole after the sudden change in its length/mass. Figure 16a shows the response of the system to an angle displacement of the 2.5 meter pole without the DHP training turned on; note the unstable oscillations resulting in a pole drop. Figure 16b shows the same test, but this time with the DHP learning turned on; note that the adaptive redesign of the controller succeeds in making its modifications without the pole falling.

INSERT FIGURE 16 ABOUT HERE

The DHP method of controller design has shown itself capable of developing controllers that are robust relative to rather large variations in plant parameters on this pole-cart testbed (and replicated in the next section). In addition, even when the plant parameters move outside this region of robustness, the DHP method demonstrated an ability to refine the controller design on line -- i.e., to perform an adaptive process -- as illustrated in Figure 16b. As reported in [Lendaris & Shannon, 1998] these are potentially very important attributes of the method, and therefore will require testing on substantially more complex plants than the pole-cart context. The next section is one such foray into more complex plants.

4.2 Narendra benchmark example.

This plant is a non-linear multiple-input-multiple-output discrete time map. It was proposed and explored in [Narendra and Mukhopadhyay, 1994]. Our treatment with DHP follows the general line used in [Visnevski and Prokhorov, 1996]. The plant has three state variables and two controls. The state equations are:

$$x_1(t+1) = 0.9x_1(t)\sin[x_2(t)] + \left[2 + 1.5\frac{x_1(t)u_1(t)}{1 + x_1^2(t)u_1^2(t)}\right]u_1(t) + \left[x_1(t) + \frac{2x_1(t)}{1 + x_1^2(t)}\right]u_2(t)$$

$$x_2(t+1) = x_3[1 + \sin[4x_3(t)]] + \frac{x_3(t)}{1 + x_3^2(t)}$$

$$x_3(t+1) = [3 + \sin[2x_1(t)]]u_2(t)$$

The observable states of the system are defined to be $x_1(t)$ and $x_2(t)$. As is often noted in the literature, this plant is stable at the origin with constant control values, but highly unstable otherwise. The linearized system at the origin is controllable, observable and of minimum phase. For the purpose of focusing on implementing DHP as cleanly as possible in this example, we choose an implementation wherein all the state variables are accessible, and all the state equations are known. For other treatments of this system, see [Narendra & Mukhopadhyay, 1994] and [Prokhorov, 1997]

The proposed control objective is to track a reference input. The benchmark reference signal proposed in [Narendra & Mukhopadhyay, 1994] for controller performance is

$$\tilde{x}_1(t) = 0.75\sin\left[\frac{2\pi t}{50}\right] + 0.75\sin\left[\frac{2\pi t}{10}\right]$$

$$\tilde{x}_2(t) = 0.75\sin\left[\frac{2\pi t}{30}\right] + 0.75\sin\left[\frac{2\pi t}{20}\right]$$

We note that this signal is periodic which period 300.

4.2.1 Defining the utility function.

For this system we use the following primary utility function:

$$U(t) = [x_1(t+1) - \tilde{x}_1(t+1)]^2 + [x_2(t+2) - \tilde{x}_2(t+2)]^2$$

Other treatments of this benchmark problem in the literature [Visnevski & Prokhorov, 1996] use a more elaborate primary utility function that introduces further time delays into the training process. While the more complex utility function works fine, our simpler implementation produces essentially the same results with less computational overhead. [Indeed, this is an example of a suggestion made previously in this chapter that there is often substantial benefit to paring down *U(t)* to contain the minimum number of terms necessary to accomplish the task (what these are, however, are not always easy to determine *a priori*).]

4.2.2 Controller implementation.

A basic controller implementation, assuming accessibility of all the state variables, has five inputs and two outputs; we endow it with six hidden elements. The inputs are the three state variables of the plant: $x_1(t), x_2(t), x_3(t)$, along with the next target values $\tilde{x}_1(t+1)$ and $\tilde{x}_2(t+2)$. The outputs are the $u_1(t)$ and $u_2(t)$. All the processing elements have hyperbolic-tangent activation functions and include bias terms.

Scaling factors selected for the state variables are as follows:

$x_1(t)$: 1.6

$x_2(t)$: 1.6

$x_3(t)$: 4.0

The controller outputs are not scaled.

4.2.3 Critic implementation.

The basic critic network has four inputs $x_1(t+1)$, $x_2(t+1)$, $\tilde{x}_1(t+1)$, and $\tilde{x}_2(t+2)$; and two outputs $\lambda_1(t) = \frac{\partial}{\partial x_1(t+1)}J(t)$ and $\lambda_2(t) = \frac{\partial}{\partial x_2(t+2)}J(t)$. Again, we use 6 hidden layer elements with hyperbolic-tangent activation functions, and include a bias term in each element. The scaling factors indicated above for the controller are used to scale the critic inputs.

4.2.4 Model implementation.

We again use an analytic model:

$$\frac{\partial}{\partial \boldsymbol{X}(t)}\boldsymbol{X}(t+1) = \begin{bmatrix} \frac{\partial x_1(t+1)}{\partial x_1(t)} & \frac{\partial x_1(t+1)}{\partial x_2(t)} & \frac{\partial x_1(t+1)}{\partial x_3(t)} \\ \frac{\partial x_2(t+1)}{\partial x_1(t)} & \frac{\partial x_2(t+1)}{\partial x_2(t)} & \frac{\partial x_2(t+1)}{\partial x_3(t)} \\ \frac{\partial x_3(t+1)}{\partial x_1(t)} & \frac{\partial x_3(t+1)}{\partial x_2(t)} & \frac{\partial x_3(t+1)}{\partial x_3(t)} \end{bmatrix}$$

with

$$\frac{\partial x_1(t+1)}{\partial x_1(t)} = 0.9\sin x_2(t) + \left[1 + \frac{2(1 - x_1^2(t))}{(1 + x_1^2(t))^2}\right]u_2(t) + 1.5\left[\frac{1 - x_1^2(t)u_1^2(t)}{(1 + x_1^2(t)u_1^2(t))^2}\right]u_1^2(t)$$

$$\frac{\partial x_1(t+1)}{\partial x_2(t)} = 0.9x_1(t)\cos x_2(t)$$

$$\frac{\partial x_2(t+1)}{\partial x_3(t)} = 1 + \sin 4x_3(t) + 4x_3(t)\cos 4x_3(t) + \frac{1 - x_3^2(t)}{(1 + x_3^2(t))^2}$$

$$\frac{\partial x_3(t+1)}{\partial x_1(t)} = 2u(t)\cos 2x_1(t)$$

and all other terms of the Jacobian are zero, and

$$\frac{\partial}{\partial \boldsymbol{u}(t)}\boldsymbol{X}(t+1) = \begin{bmatrix} \frac{\partial x_1(t+1)}{\partial u_1(t)} & \frac{\partial x_1(t+1)}{\partial u_2(t)} \\ \frac{\partial x_2(t+1)}{\partial u_1(t)} & \frac{\partial x_2(t+1)}{\partial u_2(t)} \\ \frac{\partial x_3(t+1)}{\partial u_1(t)} & \frac{\partial x_3(t+1)}{\partial u_2(t)} \end{bmatrix}$$

$$= \begin{bmatrix} 2 + \frac{3}{(1 + u_1^2(t)x_1^2(t))^2} & x(t)\left[1 + \frac{2}{1 + x_1^2(t)}\right] \\ 0 & 0 \\ 0 & 3 + \sin 2x_1(t) \end{bmatrix}$$

4.2.5 Training strategies.

1) Training for this system is complicated by the necessary delay in evaluating the quality of the controller’s actions, due to delays in the plant, and delays incorporated in U(t). The plant must be allowed to evolve two time steps into the future before all the critics inputs are available. So we resign ourselves to always correcting errors that are two or three time steps old at the time we make weight updates. After waiting two time steps at the beginning of the whole training process for enough history to be generated, the training can begin. We can once again do a simple straight forward implementation of our selected strategy (cf. Sections 2.4.1 and 4.1.5). We only have to wait once at the very beginning; after that, there is always sufficient historical data available for the algorithm to work with. The actual implementation looks very much like that for the pole-cart problem with the exception of the time subscripts.

2) The weights for both the controller and the critic networks are updated using a basic gradient descent method. Momentum terms are used in both cases with a coefficient of 0.015. No derivative offset value is used in either of the weight update processes. Once again a variety of controller and critic learning rates have been used. The results we report here are based on con-

troller and critic learning rates of 0.003 and 0.01.

3) Training stimulus is provided by a random reference signal. This signal is generated by picking values for each target uniformly from the interval [-1.5, 1.5] and holding these values fixed for four time steps. Thus the target jumps to a random pair of values, dwells there for four time steps and then jumps again. This is equivalent to upsampling a random signal by a factor of four using zero order interpolation. Such a scheme for generating training stimuli is motivated by the desire to have excitations across the range of possible system states and targets, while at the same time giving the controller a chance to iteratively refine itself (for 4 time steps) at a specific excitation before (randomly) moving on.

4) Training is performed for a total of 20,000 time steps after which all adaptation is halted (apparently this is a factor of 5 to 20 fewer steps than reported in the other cited literature), and the controller network's ability to track the benchmark sinusoidal reference signal given above is tested.

4.2.6 Simulation results.

An illustration of testing a controller trained in the above manner on the sinusoidal benchmark signal is shown in Figure 17. Note that no adaptation is taking place during this test, and that the controller has never been trained on this signal (recall, it was trained on a *random* sequence of excitations). The control quality is equivalent to that reported in [Visnevski & Prokhorov, 1996] using DHP, and to that reported in [Narendra & Mukhopadhyay, 1994] using a non-critic approach. While the example we show here was trained using a Classical training strategy, equivalent results were obtained using Shadow and Flip-Flop strategies (though the Flip-Flop strategy takes at least twice as long).

INSERT FIGURE 17 ABOUT HERE

Further generalization tests on other reference signals suggested in [Narendra & Mukhopadhyay, 1994] are illustrated in Figure 18a and Figure 18b. The second benchmark test signal is

$$\tilde{x}_1(t) = 0$$
$$\tilde{x}_2(t) = 1.5$$

and the third is

$$\tilde{x}_1(t) \sim uniform[-1.5, 1.5]$$
$$\tilde{x}_2(t) = 0$$

INSERT FIGURE 18 ABOUT HERE

No tests on these benchmark test signals are reported in [Visnevski & Prokhorov, 1996], and

[Narendra & Mukhopadhyay, 1994] report only results from a rather unsatisfactory linear controller and from their very *best* nonlinear controller, which is an order of magnitude more complex than ours and was trained for ten times as long. Even so the controller described above compares rather well with theirs.

4.2.7 On-line adaptation.

How much better does the above DHP controller be able to do if permitted to adapt on line? Figure 19 illustrates the same controller pictured in the previous examples, this time learning *on-line* to track the second benchmark signal (compare with Figure 18a). Fewer than 100 time steps into the test, the adaptation process has reduced the average RMS error from 0.24 to 0.05. Furthermore, when this adapted controller is tested on the sinusoidal reference signal, its performance is not degraded. When the same base controller is allowed to adapt to the third benchmark signal, its performance also improves significantly (RMSE drops from 0.25 to 0.13 – not pictured). As with the pole-cart experiments, we found no other reported data about this potentially *very* important application aspect of the DHP method.

INSERT FIGURE 19 ABOUT HERE

We next explored having the DHP perform the sinusoidal benchmark test with the learning turned on. When using the usual learning rates as in the original learning (and in all the other results given above), not much improvement occurred. However, when we reduced the learning rates for both the critic and controller networks to 0.001, controller performance proceeded to slowly improve. Figure 20 shows a test of the controller on the sinusoidal benchmark test after it has been adapting on-line (all the while successfully controlling the plant) for 150,000 time steps; average RMS error has dropped *monotonically* from 0.25 to 0.18 (see Figure 21), and the maximum error is much less.

INSERT FIGURE 20 & 21 ABOUT HERE

5.0 Recommendations

The following Questions/Answers were offered in the Conclusion section of [Lendaris & Shannon, 1998], and the authors feel they bear repeating here:

What strategy to use? All the strategies are capable of producing good controllers, but Classical and Shaddow Strategies do so much faster. Therefore, we suggest using Classical or Shaddow unless something in your problem context suggests Flip/Flop might be more successful.

What NN architecture to use? Start with the smallest architectures that you can conceive of, and slowly “grow” them until performance drops off. You will tend to get faster learning and better control, up to a point, after which performance diminishes.

What controller sampling rate to use? As fast a sampling rate as is convenient and/or computationally feasible. It doesn't need to be faster, but if it is, training is faster and quality of control tends to be better.

How good of a plant representation is needed in the DHP method? The plant representation doesn't need to be perfect. For example, coarse/fast integration routines with significant error are ok to use in training. Also, controllers seem to be robust with respect to plant parameter variations, thus also implying that the model doesn't need to be exact.

Is on-line learning a realistic possibility? An example was cited wherein the answer is yes.

What kind of generalization is possible? Results of tests in the Pole-Cart problem for both the θ-only and the θ-x problems, using significant θ displacements and x displacements show impressive generalization capability of the controllers designed via the DHP process, at least for the pole-cart platform. A sense was developed that even though the controller was designed to balance the pole vertically (θ=0), the same controller could then be used in a tracking problem context. If so, then good generalization in a higher sense is also accomplished.

The following are offered from the pragmatic point of view of the "nitty and gritty" of developing a designed controller. Generally speaking, make sure the simulation of the plant is accurate and well understood.

1.) Be able to separate approximations made in the simulation process from imprecise or imperfect knowledge of the plant's actual dynamics. Discrete time approximations of continuous systems should be well thought out. Recognize that computer simulation is, fundamentally, just an approximation. What is being approximated in your case? How is this approximation going to interact with the DHP process?

2.) Take care that the chain rule is correctly applied to the plant representation used. Apply the chain rule to the true system equations, not to any of the approximations thereof made during the simulation process. Take great care in calculating long chains of derivatives across several time steps. The dual NN of the Backpropagation method can be of great assistance here.

3.) Don't assume that training stimuli generated by measuring behavior about a desired reference region will lead to efficient, or even effective, training. In general, "persistent excitation" across the full range of system and reference trajectories is needed for effective learning. A trained controller can always be refined on-line after the system dynamics have been learned.

4.) Only use as many input variables as needed and no more. This is usually easier said than done. Identify what information is needed for meeting the control objectives as expressed by the selected primary utility function. Provide only that data to the controller which is needed to accomplish these control objectives, or which make the control task demonstrably easier. Identify what information is needed for evaluating the quality of the controller's actions and supply only associated data to the critic. *Be ruthless*: don't let the NNs fail in their task because they are overloaded with irrelevant or redundant variables. Do add degrees of freedom to NNs to enable better function approximation; do *not* add degrees of freedom just to perform data filtering tasks you as the designer can accomplish with a little thought.

5.) Be prepared to apply a broad range of NN training techniques and architectural elements. For example, the use or exclusion of bias terms can make a substantial difference in the training and performance characteristics of the controller. Appropriate scaling values can make a significant difference in initial controller training. Be prepared to tinker.

All in all, the authors feel there is substantial promise for the Adaptive Critic methods, and in particular the DHP variety, for application in the controller design context. Particularly enticing at this juncture is application of the ‘off-line learning for on-line adaptation’ way of using the DHP method. The preliminary results on the DHP method’s capability to perform rapid on-line refinement of the controller design in response to (even substantial) changes in its evironment beg for further study and application.

REFERENCES

Almeida, L.B., 1987, “A Learning Rule for Asynchronous Perceptrons with Feedback in a Combinatorial Environment,” *Proceedings IEEE First International Conference on Neural Networks*, San Diego, IEEE Press, June.

Athans, M. & P.L. Falb, 1966, *Optimal Control Theory*, McGraw Hill.

Barto, A.G., R.S. Sutton, & C.W. Anderson, 1983, “Neuronlike Elements That Can Solve Difficult Learning Control Problems,” *IEEE Transactions on Systems Man & Cybernectics*, vol 13, pp 834-846.

Bellman, R.E., 1957, *Dynamic Programming*, Princeton Univ. Press.

Bertsekas, D.P., 1987, *Dynamic Programming: Deterministic and Stochasic Models*, Prentice-Hall.

Bertsekas, D.P. & J.N. Tsitsiklis, 1996, *Neuro-Dynamic Programming*, Athena Scientific.

Biega, V. & S.N. Balakrishnan,1996, “A Dual Neural Network Architecture for Linear and Nonlinear Control of Inverted Pendulum on a Cart,” *Proceedings of 1996 IEEE International Conference on Control Applications*, Dearborn, MI, Sept.

Bryson, Jr. & Y. Ho, 1969, *Applied Optimal Control*, Blaisdell Publishing.

Cohen, P.R. & E.A. Feigenbaum, 1982, *The Handbook of Artificial Intelligence*, vol 3, Kauffman.

Cox, C. & R. Saeks, 1998, “Adaptive Critic Control and Functional Link Neural Networks”, to appear, *Proceedings of IEEE SMC Conference*, San Diego.

Feldkamp, L., G. Puskorius & D. Prokhorov, 1997, “Unified Formulation for Training Recurrent Networks with Derivative Adaptive Critics,” in *PROC. International Conference on Neural Networks -ICNN’97* in Houston, TX, IEEE Press.

Haykin, S., 1994, *Neural Networks: A Comprehensive Foundation*, Macmillan.

Howard, R., 1960, *Dynamic Programming and Markov Processes*, MIT Press.

Hrycej, T., 1997, *Neurocontrol: Toward An Industrial Control Methodology*, Wiley.

Lendaris,G. and C. Paintz, C., 1997, “Training Strategies for Critic and Action Neural Nets in Dual Heuristic Programming Method”, in *PROC of International Conference on Neural Networks --ICNN’97*, Houston, IEEE, pp712-717.

Lendaris,G., C. Paintz, and T. Shannon, 1997, “More on Training Strategies for Critic and Action Neural Nets in Dual Heuristic Programming Method”, in *PROC of IEEE-SMC’97*, Orlando, IEEE, October.

Lendaris, G. and T. Shannon, 1998, “Application Considerations for the DHP Methodology,” in *PROC of International Joint Conference on Neural Networks--IJCNN’98*, Anchorage, IEEE.

Miller, W.T. III, R.S. Sutton, & P.J. Werbos, eds.,1990, *Neural Networks For Control*, MIT Press.

Narendra, K.S. & P. Parthasarathy, 1990, “Identification and Control of Dynamical Systems Using Neural Networks,” *IEEE Transactions On Neural Networks*, vol 1(1), pp 4-27.

Narendra, K.S. & S. Mukhopadhyay, 1994, “Adaptive Control of Nonlinear Multivariable Systems Using Neural Networks,” *Neural Networks*, vol 7(5), pp 737-752.

Narendra,K.S., 1996, “Neural Networks for Control: Theory and Practice,” *Proceedings Of IEEE*, vol 10(2), pp 18-23.

Prokhorov, R., 1997, Adaptive Critic Designs and their Applications, Ph.D. Dissertation, Department of Electrical Engineering, Texas Tech University

Prokhorov, D., R. Santiago & D. Wunch, 1995, “Adaptive Critic Designs: A Case Study for Neurocontrol,” *Neural Networks*, vol 8(9), pp 1367-1372.

Prokhorov, D. & D. Wunsch, 1996, “Advanced Adaptive Critic Designs,” *Proceedings World Congress On Neural Networks --WCNN’96* in San Diego, CA, pp 83-87, Erlbaum.

Prokhorov, D. & D. Wunsch, 1997, “Adaptive Critic Designs,” *IEEE Transactions On Neural Networks*, vol 8(5), pp 997-1007

Rosenblatt, F., 1962, *Principles Of Neurodynamics*, Spartan Books.

Rummelhart, D.E.,& J.L. McClelland, eds., 1986, *Parallel Distributed Processing: Explorations In The Microstructures Of Cognition*, vol 1, MIT Press.

Rummelhart, D.E., G.E. Hinton, & R.J. Williams, 1986, “Learning Internal Representations by Error Backpropagation,” in [Rummelhart & McClelland, 1986], vol 1, pp 318-362.

Samuel, A.L., 1959, “Some Studies in Machine Learning Using the Game of Checkers,” *IBM Journal Of Research & Development*, vol 3, pp 210-229.

Saini, G. & S.N. Balakrishnan, 1997, “Adaptive Critic Based Neurocontroller for Autolanding of Aircraft with Varying Glideslopes,” *Proceedings International Conference On Neural Networks --ICNN’97* Houston, TX, IEEE Press.

Santiago, R. & P.J. Werbos, 1994, “A New Progress Towards Truly Brain-Like Control,” *Proceedings World Congress On Neural Networks -- WCNN’94* in San Diego, CA, pp I: 27-33.

Sutton, R.S., 1984, Temporal Aspects of Credit Assignment in Reinforcement Learning, Ph.D. dissertation, University of Massachusetts.

Visnevski, N. & D. Prokhorov, 1996, “Control of a Nonlinear Multivariable System with Adaptive Critic Designs,” in C.Dagli, et al, eds., Intelligent Engineering Systems Through Artificial Neural Networks, *PROC Conference Artificial Neural Networks In Engineering -- ANNIE’9*6, vol 6, pp 559-56, ASME Press.

Werbos, P.J., 1974, “Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences,” Ph.D. Dissertation, Committee on Mathematics, Harvard University.

Werbos, P.J., 1990a, “Backpropagation Through Time: What It Is and How To Do It,” *PROC IEEE*, vol 78(10), pp 1550-1560.

Werbos, P.J.,1990b, “A Menu of Designs for Reinforcement Learning Over Time,” Chapter 3 in [Miller, et al, 1990].

Werbos, P.J., 1992, “Approximate Dynamic Programming for Real-Time Control and Neural Modeling,” Chapter 13 in [White & Sofge, 1992].

Werbos, P.J., T. McEvoy & T. Su, 1992, “Neural Networks, System Identification, and Control in the Chemical Process Industries,” Chapter 10 in [White & Sofge, 1992].

White, D.A. & D.A. Sofge, eds., 1992, *Handbook Of Intelligent Control: Neural, Fuzzy, And Adaptive Approaches*, Van Nostrand Reinhold.

Widrow, B. and M.E. Hoff, Jr., 1960, “Adaptive Switching Circuits,” *IRE WESCON Convention Record*, pp 96-104.

Widrow, B., N. Gupta & S. Maitra, 1973,”Punish/Reward: Learning With a Critic in Adaptive Threshold Systems,” *IEEE Transactions On Systems Man & Cybernetics*, vol 3(5), pp 455-465.

FIGURE TITLES

during training, (b) average accumulated error during testing in 100 trials.

Figure 13:

Comparison of training and performance quality of different θ-x controllers; (a) average drops during training, (b) average accumulated angular error, and (c) track position error during testing in 100 trials.

Figure 14:

Comparison of training and performance quality of θ-only controllers over a range of sampling intervals; (a) average drops during training, (b) average accumulated cost during testing in 100 trials.

Figure 15:

Comparison of training and performance quality of different θ-x controllers over a range of sampling intervals; (a) average drops during training, (b)average accumulated angular error, and (c) track position error during testing in 100 trials.

Figure 16:

Step response of a 6-1-1 controller trained using a 1m pole being tested on a 10° displacement using a **2.5m** pole; (a) without on-line learning, the controller is unable to balance the longer pole, (b) with on-line learning the controller adapts to the new pole without dropping. Solid/dashed lines are the pole angle/track position.

Figure 17:

Performance of a DHP controller on the Narendra sinusoidal benchmark test signal. Solid/dashed lines are the actual/desired trajectories.

Figure 18:

Performance of a DHP controller on other Narendra benchmark test signals; (a) $r_1(t) = 0$, $r_2(t)=1.5$, (b) $r_1(t) \sim$ uniform[-1.5, 1.5], $r_2(t) = 0$. Solid/dashed lines are the actual/desired trajectories.

Figure 19:

Performance of a DHP controller with on-line learning on the Narendra benchmark signal $r_1(t) = 0$, $r_2(t)=1.5$ (compare with Figure 18a). Solid/dashed lines are the actual/desired trajectories.

Figure 20:

Average RMS error over time for a DHP controller adapting on-line while tracking the Narendra sinusoidal test signal.

Figure 21:

Performance of a DHP controller after 150,000 steps of on-line adaptation on the Narendra sinusoidal test signal (compare with Figure 17). Solid/dashed lines are the actual/desired trajectories.

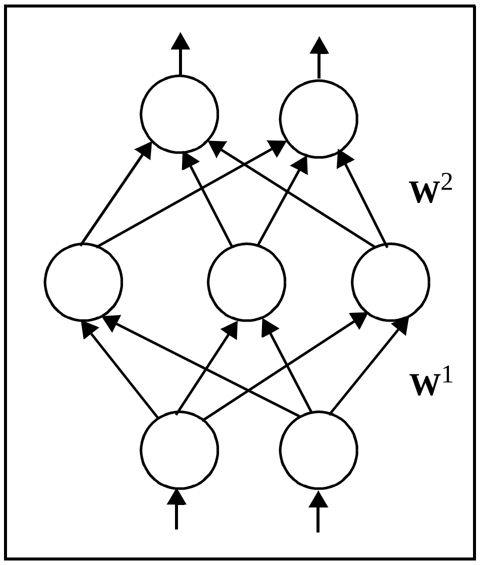

Figure 1Feed-forward Multi-layer Neural Network (shown with 1 hidden layer).

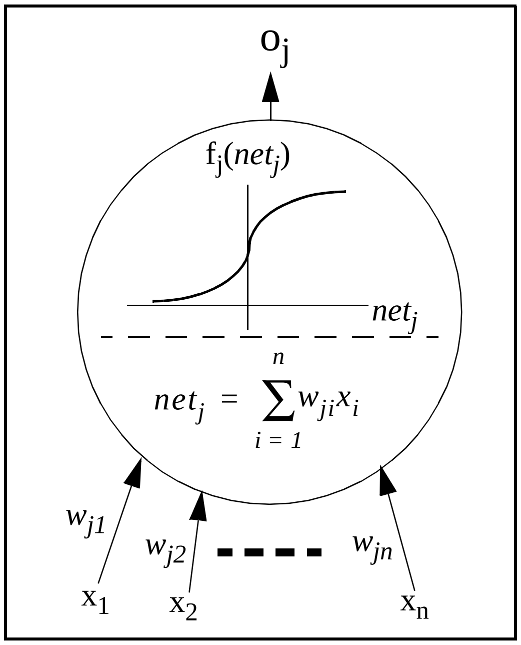

Figure 2Generic (*j*th) Neural Element

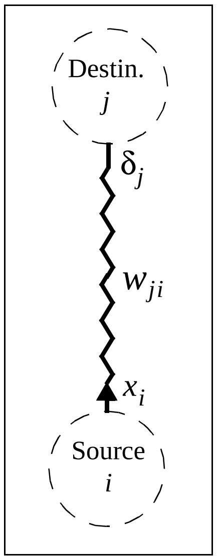

Figure 3Generic weight with signal at its source side x_i, and a *proxy* signal value δ_j at its destination side representing some attribute of the destination-neural-element's activity.

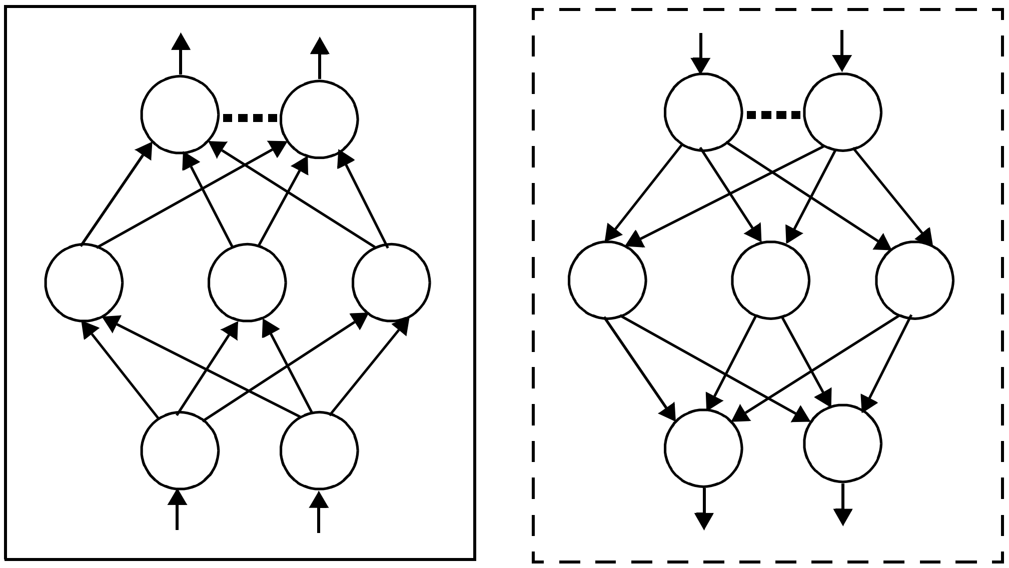

Figure 4Feed-forward NN being trained (left); corresponding *dual NN (right).*

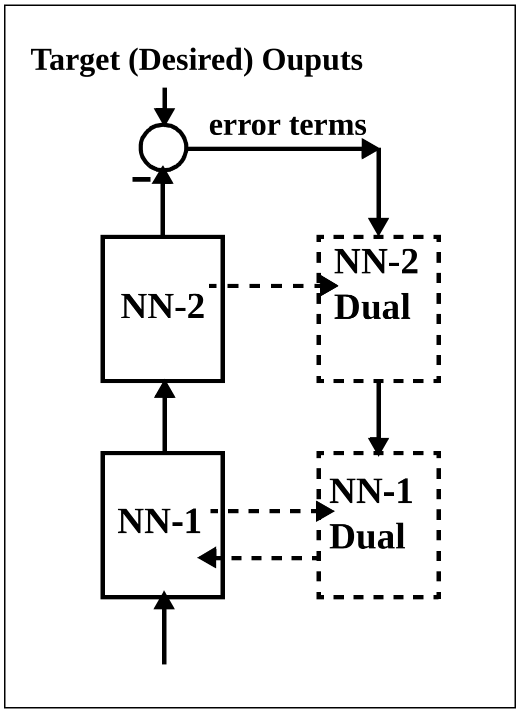

Figure 5Use of dual NNs to obtain δ_j values for training NN-1 based on error information available at output of NN-2.

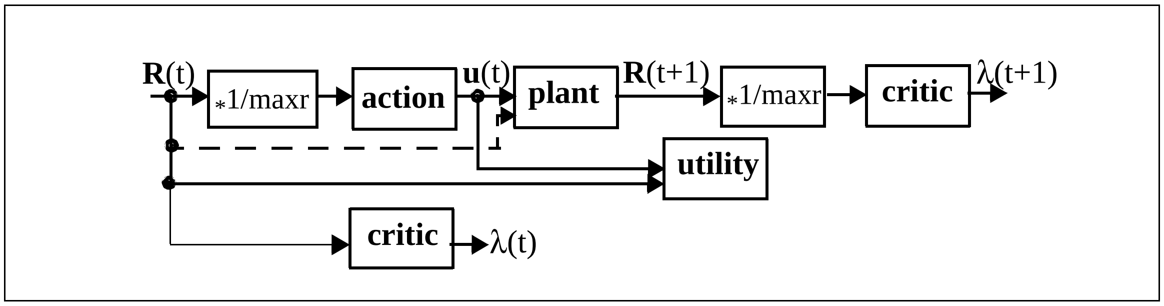

Figure 6 Schematic Diagram of important components of DHP process.

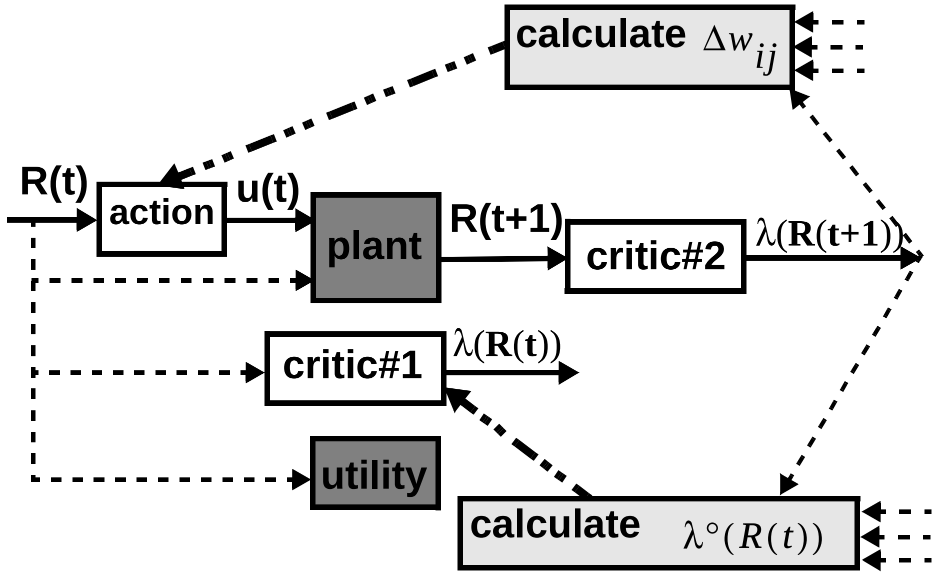
calculate Δw_{ij}
R(t)
action
u(t)
plant
R(t+1)
critic#2
$\lambda(\mathbf{R(t+1)})$
critic#1
$\lambda(\mathbf{R(t)})$
utility
calculate $\lambda°(R(t))$

Figure 7Computing Schema for Discussing Strategies.

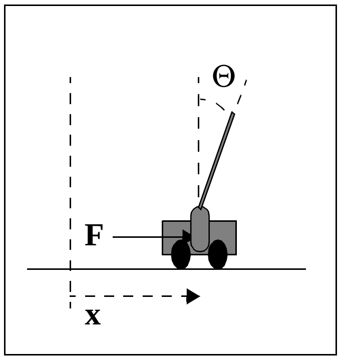

Figure 8 Schematic of Pole-Cart testbed problem.

Figure 9

Pole angle in radians during the first 30 seconds of training on a 5° angle (a) and during a test of the trained controller on a 38° angle (b). Architecture is 3-1-1, 3-1-3 learning the θ-only problem using a classical update strategy.

Figure 10

Step responses during generalization testing of a 6-1-1 θ-x controller for (a) a 38° displacement and (b) a -6.6m displacement of the cart. Solid/dashed lines are the pole angle/track position.

Figure 11

Comparison of step responses for a 7.5° displacement of (a) 4-1-1 controller, (b) 6-1-1 controller, (c) 6-3-1 controller, and for a 1.5m displacement (d), (e) and (f). Solid/dashed lines are the pole angle/track position.

Figure 12

Comparison of training and performance quality of different θ-only controllers; (a) average drops during training, (b) average accumulated error during testing in 100 trials.

Figure 13

Comparison of training and performance quality of different θ-x controllers; (a) average drops during training, (b) average accumulated angular error, and (c) track position error during testing in 100 trials.

Figure 14

Comparison of training and performance quality of θ-only controllers over a range of sampling intervals; (a) average drops during training, (b) average accumulated cost during testing in 100 trials.

Figure 15

Comparison of training and performance quality of different θ-x controllers over a range of sampling intervals; (a) average drops during training, (b)average accumulated angular error, and (c) track position error during testing in 100 trials.

Figure 16

Step response of a 6-1-1 controller trained using a 1m pole being tested on a 10° displacement using a **2.5m** pole; (a) without on-line learning, the controller is unable to balance the longer pole, (b) with on-line learning the controller adapts to the new pole without dropping. Solid/dashed lines are the pole angle/track position.

Figure 17

Performance of a DHP controller on the Narendra sinusoidal benchmark test signal. Solid/dashed lines are the actual/desired trajectories.

Figure 18

Performance of a DHP controller on other Narendra benchmark test signals; (a) $r_1(t) = 0$, $r_2(t)=1.5$, (b) $r_1(t) \sim$ uniform[-1.5, 1.5], $r_2(t) = 0$. Solid/dashed lines are the actual/desired trajectories.

Figure 19

Performance of a DHP controller with on-line learning on the Narendra benchmark signal $r_1(t) = 0$, $r_2(t)=1.5$ (compare with Figure 18a). Solid/dashed lines are the actual/desired trajectories.

Figure 20

Average RMS error over time for a DHP controller adapting on-line while tracking the Narendra sinusoidal test signal.

Figure 21

Performance of a DHP controller after 150,000 steps of on-line adaptation on the Narendra sinusoidal test signal (compare with Figure 17). Solid/dashed lines are the actual/desired trajectories.