

Application Considerations for the DHP Methodology

George G. Lendaris¹, Thaddeus Shannon²

1. Professor, Systems Science & Electrical Engineering, lendaris@sycs.pdx.edu

2. Graduate Student, Systems Science Ph.D. Program
Portland State University
PO Box 751, Portland, OR 97207

Abstract

Six questions are posed that are likely to be of interest to one considering applying the DHP methodology. Tentative answers to these questions are provided in the last section of the paper. As a basis for these, the paper first describes observations made from a series of explorations into various aspects of using the DHP method to design the controller for the pole-cart benchmark problem. Parameters of the explorations included training gains, plant parameter variations, fidelity of plant model, controller sampling rates, network architectures, training strategies, and generalization tests.

1. Introduction

Dual Heuristic Programming (DHP) is emerging as a potentially very useful and powerful method for designing controllers. The present paper aspires to assist those contemplating the possible use of the DHP in their application. A set of questions a potential user is likely to ask include the following: What training strategy do I use to perform DHP? What NN architecture is best? What controller sampling rate should be used? How good of a plant representation is needed? Is on-line learning a realistic possibility? What kind of generalization is possible?

The platform used for the explorations reported herein was the pole-cart benchmark problem [1][2][3]. This of course represents a relatively simple plant, but nevertheless, the authors feel the results obtained have some application value.

2. Mathematical foundations

The Dual Heuristic Programming (DHP) method is a neural network approach to solving the Bellman equation [9]. The latter entails maximizing a specified (*secondary*) utility function:

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

The term γ^k is a discount factor ($0 < \gamma < 1$) [assumed to be 1 in this paper] and $U(t)$ is the *primary* utility function, defined by the user for the specific application context. A useful identity:

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

The usual application context for the DHP method is control [9]. It is useful to observe that the Bellman-type optimization going on in the DHP method refers to the

process of **designing** the controller (*actionNN* herein).

Two methods for discretizing continuous models of plants are used -- see Section 5.1. Two neural nets are used, one for the *actionNN* functioning as the controller, and one for the *criticNN* used to design (via training) the *actionNN*. A third NN could be trained separately to copy the plant if an analytical description (model) for it is not available.

$\mathbf{R}(t)$ [dim. n] is the state of the plant at time t. The control signal $\mathbf{u}(t)$ [dim. a] is generated by the *actionNN* in response to the input $\mathbf{R}(t)$. The signal $\mathbf{u}(t)$ is then asserted to the plant. As a result of this, the plant changes its state to $\mathbf{R}(t+1)$. The *criticNN*'s role is to assist in developing/designing a controller (*actionNN*) that is "good" relative to optimizing the specified utility function $U(\mathbf{R}(t), \mathbf{u}(t))$, which is crafted to express the objective and constraints of the control application. In the DHP method, the *criticNN* estimates the gradient of $J(t)$ with respect to $\mathbf{R}(t)$; the letter λ is used as a short-hand notation for this gradient, so the output of the *criticNN* is designated λ .

3. Updating Process

We refer to Figure 1 to describe the computational steps used in the DHP methodology. The boxes with dark shading mean we have an analytical expression for that item [note: 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 training process. The dotted lines represent calculated values being fed into the respective boxes. The heavier dot-dash lines indicate where the learning/updating process occurs.

Reading Figure 1 from left to right, the current state $\mathbf{R}(t)$ is fed to the *actionNN*, which then generates $\mathbf{u}(t)$. The model is informed of $\mathbf{R}(t)$ and $\mathbf{u}(t)$, and then generates $\mathbf{R}(t+1)$. $\mathbf{R}(t)$ is also fed to the *critic#1* box and to the utility box, which generate, respectively, $\lambda(\mathbf{R}(t))$ and $U(\mathbf{R}(t))$. [Note, in the examples used herein, the crafted Utility functions make use of $\mathbf{R}(t)$ but not $\mathbf{u}(t)$; if $\mathbf{u}(t)$ is included in the definition of the utility U , then we would show a connection from the output of the *actionNN* to the utility box.] After the model generates $\mathbf{R}(t+1)$, this is fed to the *critic#2* box, which then yields $\lambda(\mathbf{R}(t+1))$ -- more simply denoted as $\lambda(t+1)$. 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 *actionNN* and the *criticNN*. The upper medium-shaded box calculates Δw_{ij} , and the lower medium-shaded box calculates $\lambda^\circ(t)$, the "desired" or "target" value for $\lambda(t)$ to enable

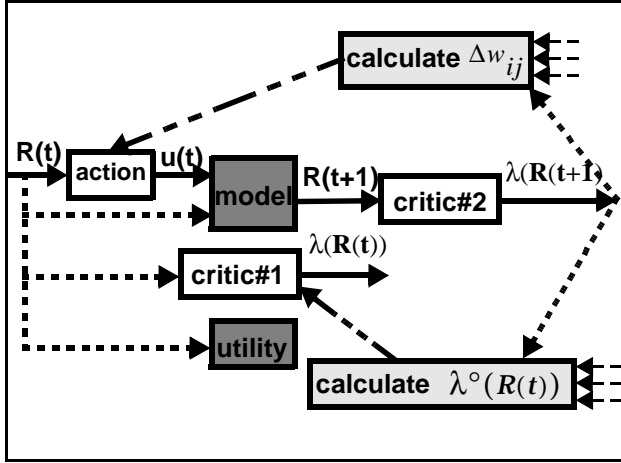


Figure 1. Computing Schema for Discussing Strategies (backpropagation-type) training of critic#1.

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

3.1 The upper medium-shaded box.

In the present work, a basic Backpropagation algorithm is used (no embellishments) to adjust the weights in the action box. The weight-adjustment is calculated via:

$$\Delta w_{ij}(t) = lcoef \cdot \frac{\partial}{\partial w_{ij}(t)} J(t) \quad (3)$$

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

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, $\frac{\partial}{\partial u_k(t)} J(t+1) = \sum_{s=1}^n \frac{\partial}{\partial R_s(t+1)} J(t+1) \cdot \frac{\partial}{\partial u_k(t)} R_s(t+1)$ (4)

Abbreviation: $\frac{\partial}{\partial R_s(t+1)} J(t+1) = \lambda(t+1)$ (5)

$\lambda(t+1)$ is approximated by the critic, in response to the input $\mathbf{R}(t+1)$.

$\frac{\partial}{\partial u_k(t)} R_s(t+1)$ can be calculated from analytical equations of the plant, if they are available, or by backpropagation through a third neural net that has been previously trained to copy the plant.

3.2 The lower medium-shaded box.

In [2], Equation (6), which makes use of the identity of Equation (2), is shown to hold. Note again the term containing the gradient of $J(t+1)$ with respect to $\mathbf{R}(t+1)$: $\lambda(t+1)$. The subscript s indicates the components of the vector quantity $\lambda(t+1)$.

$$\lambda_s^o(t) = \frac{\partial}{\partial R_s(t)} U(t) + \sum_{j=1}^u \left(\frac{\partial}{\partial u_j(t)} U(t) \cdot \frac{\partial}{\partial R_s(t)} u_j(t) \right) \quad (6)$$

$$+ \sum_{k=1}^n \left(\frac{\partial}{\partial R_k(t+1)} J(t+1) \cdot \frac{\partial}{\partial R_s(t)} R_k(t+1) \right)$$

$$+ \sum_{k=1}^n \left\{ \sum_{j=1}^a \left(\frac{\partial}{\partial R_k(t+1)} J(t+1) \cdot \frac{\partial}{\partial u_j(t)} R_k(t+1) \cdot \frac{\partial}{\partial R_s(t)} u_j(t) \right) \right\}$$

For convenience, we paraphrase Equation (6) in Equation (7). Using this version, we note that the [\sim Action] terms are calculated via the action box in Figure 1, and the [\sim Critic(t+1)] terms are calculated via the critic#2 box. Similarly, the [\sim Plant] and the [\sim Utility] terms are calculated via the respective dark-shaded box in Figure 1.

$$\lambda_s^o(t) = [\sim\text{Utility}] + \sum_{j=1}^n ([\sim\text{Utility}] \cdot [\sim\text{Action}])$$

$$+ \sum_{k=1}^n ([\sim\text{Critic}(t+1)] \cdot [\sim\text{Plant}])$$

$$+ \sum_{k=1}^n \left\{ \sum_{j=1}^a ([\sim\text{Critic}(t+1)] \cdot [\sim\text{Plant}] \cdot [\sim\text{Action}]) \right\} \quad (7)$$

3.3 The training/update process.

Detailed descriptions of various strategies were given in [2] for "solving" (iterating) Equation (7). Using Figure 1, we describe three of them here (Strategies 1 ["classical"], 2 [Prokhorov/Santiago/Werbos/Wunch, called "flip/flop"] & 4 [Lendaris/Paintz]). Keep in mind that the output of critic#2 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 boxes.

Strategy 1. Straight application of the equation.

In Figure 1, this means that after $\lambda(t+1)$ 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 boxes labeled critic#1 and critic#2 are always maintained identical -- i.e., could be the same physical box, just used for two different calculations.]

Strategy 2. Basic 2-stage process ["flip/flop"].[5]-[9]

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

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

Strategy 4. Single-stage process (as in Strategy 1), with the modification that critic#1 and critic#2 are made **two physically distinct objects**.

After $\lambda(t+1)$ 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.

[*Strategies 2 and 4 here were 2a and 4a in [2] & [3].]

4. Overview of Explorations

In early explorations [2] [3], we found Strategy 4 to have tantalizing improvements over Strategy 2 (the ‘flip/flop’ strategy of [5] - [9]). Specifically, Strategy 4 was found able to converge at least two times faster than flip/flop. It was decided to make more extensive explorations, both of the DHP process itself and of the relative performance of the strategies. These are reported here.

The platform used for these explorations continues to be the benchmark pole-cart problem [1] [2] [3], both in its ‘Theta-Only’ version, and the ‘Theta-X’ version. The explorations focused on various aspects:

- Minimum NN architectures for successful training.
- Controller design based on low fidelity models of plant
- Robustness of resulting controller with respect to plant parameter variations.
- Sampling rate of controller vs. learning speed and quality of control.

In any controller-design context, the user’s desires/constraints are specified, and the controller is designed accordingly. In the present context, the specifications are provided in the form of a ‘utility’ function. For the Θ -only version, the utility function is defined only in terms of Θ (e.g., $U(t) = -\Theta(t)^2$). With such a utility function, no limits are placed on the position $X(t)$ of the cart. The only linkage of position to angle in this case is through the (very) small friction coefficient term in the expression for Θ . An example of a utility function that brings cart position into play would be $U(t) = -\Theta(t)^2 - X(t)^2$, yielding what is called here the $\Theta - X$ problem. Clearly, scaling coefficients can be placed in front of the component terms to emphasize angular position or track position as desired. For the Θ -only version, no position data need be input to the controller, nor to the critic NN. For the $\Theta - X$ problem, it is required that both angle and position data be supplied to the controller and the criticNN.

5. Exploration Results

5.1 Minimum NN architectures for successful training.

In [3], it was found desirable to limit the state variables at the input of the action and critic NNs to those contained in the utility function. For example, in the Θ -only context, if state variables associated with cart position X were included as inputs to the NNs in addition to the required state variables associated with pole angle Θ , the DHP process did

not always converge.

A series of experiments was carried out to discover the minimum-size *actionNN* and *criticNN* that yield successful performance of the DHP process for the pole-cart problem.

The series included two different methods for simulating the plant (pole-cart platform). One is here called the Discrete-Map version, wherein the plant-simulation integration time interval was set equal to the sampling interval of the controller (*actionNN*). In the associated computations, the dependence of the angle on the angular acceleration was omitted, and this caused certain time lags to emerge in the simulation. The second method allowed independent selection of time intervals for plant-simulation integration and sampling period of the controller. The terms missing in the Discrete-Map version were included, thus eliminating the time lags. This made the simulation more faithful to the underlying differential equation -- hence called herein the Continuous-Simulation version.

In principle, for a second-order dynamical system, only the position and velocity terms (for Θ and X) are required to specify the system. In the Discrete-Map version of the plant simulation, however, due to the absence of the acceleration terms in the plant simulation, it was discovered that the *criticNN* required an acceleration term input to it, apparently so it could ‘reconstruct’ some of the missing plant information. For the Continuous-Simulation version however, the *criticNN* performed satisfactorily without the acceleration inputs -- but did better with them.

Both the *actionNN* and *criticNN* have feedforward architecture, with one hidden layer using hyperbolic-tangent activation functions. Output layer elements for the *criticNN* have linear activation functions, while for the *actionNN*, those with sigmoid activation functions produce superior controllers to those with linear activation functions.

The *actionNN* for the pole-cart problem has just a single output. The number of inputs to the *actionNN* need only be the position and velocity terms, but superior controller performance was found by including acceleration terms also. Thus, for the Θ -only version, the minimal *actionNN* has only 2 inputs (position & velocity for Θ); for the $\Theta - X$ version, the minimal *actionNN* has 4 inputs (addition of position & velocity for X). For both the Discrete-Map and Continuous plant simulations, all three DHP strategies (1, 2 & 4) yielded the same ‘minimal’ architecture: 2-1-1 for the Θ -only problem, and 4-1-1 for the $\Theta - X$ problem (input-hidden-output layer elements).

For the *criticNN*, the results were different for the Discrete-Map and Continuous simulations. In the Discrete-Map case, an additional acceleration term was required for both Θ and X into (and hence out of) the *criticNN*. Whereas for the Continuous-Simulation case the minimal architecture was 2-1-2 for Θ -only, and 4-1-4 for $\Theta - X$, for the Discrete-Map case the additional acceleration term was required, yielding a minimal architecture of 3-1-3 and 6-1-6, respectively.

The controller designs that emerged from the DHP process using single-element hidden layer NNs performed as well, and in some cases better than, controller designs

emerging from DHP processes using NNs with more elements in the hidden layer. See Figure 2 a-c & f-h.

For the Θ -only case, the training performance measures for the DHP process improved, first after adding an acceleration input, and next, by increasing the number of hidden-layer elements from one to three. For the Θ -X case, the training performance measures also improved with the addition of acceleration inputs, but addition of hidden-layer elements produced mixed results. The more complex networks were less likely to learn to control for both angle and position.

For the Θ -X problem, the simpler controllers exhibited better generalization (fewer drops) than did the more complex. This is another example of the conjecture [4] (often repeated in different contexts/formats) that given two architectures that train up equally well on given training data, the “simpler” architecture has a higher likelihood of better generalization.

A general observation that can be made is that for both the Θ -only and the Θ -X problems, and for all architectures, Strategies 1 & 4 yielded training performance measures that were twice as good as those for Strategy 2 (flip/flop of [5]-[9]). The quality of the resulting controllers via all three strategies, with the Continuous-Simulation method for the plant, however, were on average, equivalent.

5.2 Controller design based on low fidelity models of plant.

Controllers were designed (via training) using the DHP method based on a “coarse” integration routine simulating the plant, and their performance was checked using a more accurate integration routine for the plant simulation. It was found that on average, the controller performed *better* on the more accurate simulation than on the (coarser) simulation that it had been trained on.

Typically, it is the case that a model is incomplete - or “coarse” in some sense - relative to the plant being represented. Accordingly, the above result suggests a happy conclusion that the DHP method could be used to design a controller in the “lab” based on a model, and then successfully fielded on the original plant.

5.3 Robustness of resulting controller with respect to plant parameter variations

The explorations in this and the next section were all performed using the Continuous-Simulation for the plant. The controller was designed using the DHP method, based on a sequence of angular displacements of the pole (see [2]). The resulting controller was then tested for quality of generalization on angles not included in the training set, and for the Θ -X case, was also tested on some X displacements. Figure 2d shows the step response to a 38° displacement of the pole angle for one such controller (the maximum angle used during training was 10°); note the very nice response. Figure 2i shows the step response to a substantial initial displacement in the cart’s X position for the same controller. This response is very nice, especially since the controller was not explicitly trained on displacements in X. It is interesting that the controller learned sufficient information

about the pole dynamics to start the correction of the X displacement by moving further away from the target position at the beginning, in order to get the pole moving in a way that would allow the controller to get Θ and X to rest at their respective target positions at the same time.

For the same controller (6-1-1), the length/mass of the pole was then increased in varying amounts to determine the robustness of the controller design. It was observed that the controller successfully (i.e., no drops) balanced the pole for increases in the length/mass of the pole from a starting value of 1 unit all the way up to approximately 2 units. Above 2 units, 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 2e shows the response of the system to an angle displacement of the pole without the DHP training turned on; note the unstable oscillations resulting in a pole drop. Figure 2j 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. These examples were for the case of sudden change of length/mass to 2.5 units.

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 the pole-cart testbed]. In addition, even when the plant parameters move outside this region of robustness, the DHP method may be able to refine the controller design on line -- i.e., to perform an adaptive process -- as illustrated in Figure 2j. These are potentially very important attributes of the method, and therefore will require testing on substantially more complex plants than the pole-cart context. Early results on a more complex platform (see [3]) support the above robustness conclusion; tests are yet to be made related to the on-line/adaptive aspect.

5.4 Controller sampling rate vs. learning speed and quality of control

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.

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.

5.5 Training parameter values vs. initialization regions in weight space for high probability of DHP convergence.

Ideally, one would like a proof that the DHP training algorithm converges to an acceptable controller design given some range of starting weight values and a set of training parameter values, for the class of plants and control objectives/constraints one is concerned with. Absent such a proof, it is reasonable to ask if there exists a set of training parameters and a region of initial network weight values for which the process always converges in experience. For the present plant, the answer is yes, there exists a range of training parameters and an interval in weight space over which the network weights can be randomly initialized which in practice seems to always lead to successful controller training.

This is true for both the Θ -only and the Θ -X problems using Strategies 1 and 4 with the range of architectures described above. This is also true for Strategy 2 with some of the architectures.

The range of training stimuli also impact the process. Two observations may be stated based on our experiments. First, increasing the magnitude of the training angles makes the training process more difficult and less likely to converge. Second, the usual observation that 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 task.

6. Conclusion

In the Introduction, we posed a set of questions of potential interest to a person contemplating using the DHP process for designing a controller in a given problem context. Based on the experiences described above, we offer the following answers.

What strategy to use? All the strategies are capable of producing good controllers, but Strategies 1 and 4 do so much faster. Therefore, we suggest using Strategy 1 or 4

unless something in your problem context suggests Strategy 2 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 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 ($\Theta=0$), the same controller could then be used in a tracking problem context. If so, then good generalization in a higher sense is also accomplished.

7. References

- [1] Barto, A., Sutton, R. & Anderson, C. "Neuronlike Adaptive Elements that can Solve Difficult Learning Control Problems" in *IEEE SMC Trans.*, Vol.SMC-13, No.5, Sep/Oct 1983.
- [2] Lendaris, G. and Paintz, C. "Training Strategies for Critic and Action Neural Nets in Dual Heuristic Programming Method", in *PROC of ICNN'97, Houston, IEEE*, pp712-717, June, 1997.
- [3] Lendaris, G., Paintz, C., and Shannon T. "More on Training Strategies for Critic and Action Neural Nets in Dual Heuristic Programming Method", in *PROC of IEEE-SMC'97, Orlando, IEEE*, October, 1997.
- [4] Lendaris, G. and Stanley, G., "On the Structure Dependent Properties of Adaptive Logic Networks," GM Defense Research Labs., TR63-219, Jul 1963.
- [5] Prokhorov, D. and Wunsch, D. "Advanced Adaptive Critic Designs", *PROC WCNN'96*, pp. 83-87, San Diego, Erlbaum, Sept. 1996.
- [6] Prokhorov, D., Santiago, R. & Wunsch, D., "Adaptive Critic Designs: A Case Study for Neurocontrol", in *Neural Networks*, vol. 8, no. 9, pp 1367-1372, 1995.
- [7] Santiago, R. & Werbos, P. "New Progress Towards Truly Brain-Like Intelligent Control", *PROC WCNN '94*, pp. 1-2to1-33, Erlbaum, 1994.
- [8] Visnevski, N. & Prokhorov, D. "Control of a Nonlinear Multivariable System with Adaptive Critic Designs", in *Intelligent Engineering Systems through Artificial Neural Networks 6 (PROC. ANNIE '96)*, Dagli, et.al., Eds., ASME Press, pp. 559-565, 1996.
- [9] Werbos, P. "Approximate Dynamic Programming for Real-Time Control and Neural Modeling", Ch. 13 in *Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches*, (White and Sofge, eds.), Van Nostrand Reinhold, New York, NY, 1994.