Training Strategies for Critic and Action Neural Networks in Dual Heuristic Programming Method

George G. Lendaris[1] and Christian Paintz[2]
1.Professor, Portland State University, lendaris@sysc.pdx.edu
2.Graduate Student, Electrical Engineering, Portland State University
PO Box 751, Portland, OR 97207

Abstract

This paper discusses strategies for and details of training procedures for the Dual Heuristic Programming (DHP) methoology, defined in [6]. This and other approximate dynamic programming approaches (HDP, DHP, GDHP) have been discussed in some detail in [2], [4], [5], all being members of the Adaptive Critic Design (ACD) family. The example application used is the inverted pendulum problem, as defined in [1]. This "plant" has been successfully controlled using DHP, as reported in [4]. The main recent reference on training procedures for ACDs is [2]. The present paper suggests and investigates several alternative procedures and compares their performance with respect to convergence speed and quality of resulting controller design. A promising modification is to introduce a **real** copy of the *critic*NN (*critic*NN#2) for making the "desired output" calculations, and very importantly, this ***critic*NN#2 is trained differently than is *critic*NN#1**. The idea is to provide the "desired outputs" from a stable platform during an epoch while adapting the *critic*NN#1. Then at the end of the epoch, *critic*NN#2 is made identical to the then-current adapted state of *critic*NN#1, and a new epoch starts. In this way, both the *critic*NN#1 and the *action*NN can be simultaneously trained on-line during each epoch, with a faster overall convergence than the older approach. Further, the measures used herein suggest that a "better" controller design (the *action*NN) results.

1. Dual Heuristic Programming (DHP)

DHP is a neural network approach to solving the Bellman equation [6], [3]. The idea is to maximize a specified (*secondary*) utility function $J(t)$, where $J(t)$ is defined as:

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

The term γ^k is a discount factor ($0 < \gamma < 1$), and $U(t)$ is the *primary* utility function, which must be defined by the user for the specific application context. In this paper, γ is assumed to be 1. In this case, Equation (1) is equivalent to

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

A schematic diagram of important components of the DHP method is shown in Figure1.

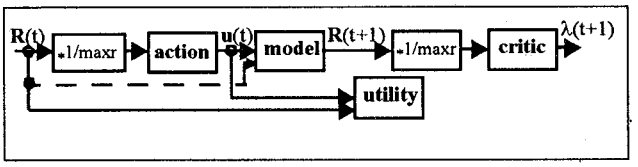


Figure 1: Schematic Diagram of important components of DHP process.

The equations and techniques described in this paper are based on a discrete plant; the usual method of discretizing continuous models of plants is used. For DHP, at least two neural nets are needed, one for the ***action*NN** functioning as the controller, and one for 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.

$\mathbf{R}(t)$ [dim: n] is the state of the plant at time t. The control signal $\mathbf{u}(t)$ [dim: a] is generated by the *action*NN 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 neural nets are updated using $\{\mathbf{u}(t), \mathbf{R}(t), \mathbf{R}(t+1)\}$in the equations described in Section 3. The *critic*NN is needed to adapt the *action*NN to the plant (model) and to the utility function. The (primary) utility function $U(\mathbf{R}(t), \mathbf{u}(t))$ expresses 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".

It was discovered useful to insert the *1/maxr modules to scale the n-dimensional state space R^n to $[-1,+1]^n$.

2. Equations to update the neural nets

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.

Equations to update the action network *weights:*

The weights in the *action*NN are updated with the objective of maximizing $J(t)$. For the NN structures defined in Section 5, the dimension of the control signal $\mathbf{u}(t)$ is $a = 1$ and a basic Backpropagation algorithm is used (no embellishments), wherein the *action*NN's weight-ad-

0-7803-4122-8/97 $10.00©1997 IEEE

justment increment is calculated via:

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

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,

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

Abbreviation:

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

$\lambda(t+1)$ is approximated by the critic, in response to the input **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.

Desired output for the critic

To train the *critic*NN, whose output is λ, a value has to be calculated for the role of "desired output", here called λ°.

Recalling Equations (1) and (5), and making use of Equation (2), we write

$$\lambda_s^\circ(t) = \frac{\partial}{\partial R_s(t)} J(t) = \frac{\partial}{\partial R_s(t)} (U(t) + J(t+1)) \qquad (6)$$

this resolves to

$$\lambda_s^\circ(t) = \frac{\mathrm{d}}{\mathrm{d}R_s(t)} U(t) + \sum_{j=1}^{a} \left(\frac{\partial}{\partial u_j(t)} U(t) \bullet \frac{\partial}{\partial R_s(t)} u_j(t) \right) \qquad (7)$$

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

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

$\frac{\mathrm{d}}{\mathrm{d}R_s(t)} U(t)$ is the (total) derivative of the primary utility function $U(t)$ with respect to $R_s(t)$.

A typical form for utility functions is as follows:

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

with constant a_i, b_i, c_j and d_j.

The partial derivative $\frac{\partial}{\partial R_s(t)} u_j(t)$ is calculated by backpropagation through the *action*NN, and

$\frac{\partial}{\partial R_k(t+1)} J(t+1)$ is approximated by the critic itself as response to $R(t+1)$, i.e., is $\lambda_k(t+1)$.

For training the *critic*NN, the "error" components are calculated as follows: $e_s = (\lambda_s^\circ(t) - \lambda_s(t))^2$ (8)

3. Strategies to update the neural nets

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

Equation (7) for $\lambda_s^\circ(t)$ is our principal focus here. [As an aside, if we substitute $\lambda_s(t)$ (i.e., without the superscript) in the left side of Equation (7), 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 (7) as follows:

$$\lambda_s^\circ(t) = [\sim Utility] + \sum_{j=1}^{a} ([\sim Utility] \bullet [\sim Action])$$

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

$$+ \sum_{k=1}^{n} \left\{ \sum_{j=1}^{a} ([\sim Critic(t+1)] \bullet [\sim Plant] \bullet [\sim Action]) \right\} \qquad (9)$$

In a neural network implementation, the *[~Action]* terms are calculated via the *action*NN, and the *[~Critic(t+1)]* terms are calculated via the *critic*NN. Various strategies could be used to "solve" (iterate) Equation (9). Several are discussed here.

Strategy 1. "Straight" application of the equation.

Strategy 2. Basic two-stage process:

i) Hold the *action*NN parameters constant for a specified number N of computation cycles, while adjusting parameters in the *critic*NN.

ii) Then, hold the *critic*NN parameters constant for a specified number M of computation cycles, while adjusting the *action*NN parameters. Return to i).

NOTE: For the purposes of this paper, we let N=M, and use the familiar term "epoch" as a name for this set of cycles.

Strategy 3. Modified two-stage process (as in **2.**, however, first stage is substantially modified):

i) We noted earlier that the task of the *critic*NN [which performs $\lambda(\mathbf{R})$] is to learn λ°. For the present strategy, we temporarily *suspend* adjustments to the process that calculates λ° to give the process that calculates λ a chance to accomplish its adaptation. To do this, we create a new copy of the *critic*NN; the original is called *critic*NN#*1* and the copy is called *critic*NN#*2*. [NOTE: This is different from the "copy" in the cited references; their "copy" is for convenience rather than substance.] In the present case, we hold *critic*NN#*2's* parameters constant during the epoch; we use this (non-adapted) *critic*NN#2 to calculate λ° and use these values of λ° to train *critic*NN#*1*. As in **2.**, the *action*NN parameters are held constant during this epoch. At the end of this epoch, the weights of *critic*NN#*2* are set equal to those of *critic*NN#*1*.

ii) Same as for **2.** above.

Strategy 4. Single-stage process, based on suspended adaptation of *critic*NN#2.

We maintain the notion of an epoch as in **2.** and **3.**. However, in this strategy, BOTH the *action*NN and the *critic*NN#1 are adjusted during each computational cycle. As before, *critic*NN#*2* is not adapted during the epoch; its weights are set equal to those of *critic*NN#1 at the end of the epoch. The next epoch starts the same process over.

Algorithms for the above strategies:

Strategies **1.** & **4.** are single-stage processes. Strategies **2.** and **3.** are 2-stage processes. The 2-stage processes are (here) said to comprise a sequence of "flip/flop" epochs. In the "flip" epoch, only the *critic*NN is updated; in the "flop" epoch, only the *action*NN is updated (on-line mode). The variations in Strategies **2.** and **3.** occur in the flip epoch; the flop epoch process is the same for both. The flip and flop epochs are here equal in length.

Flop epoch algorithm (adapts the *action*NN in on-line mode):

1. Scale **R**(t) and apply it to the *action*NN, obtain **u**(t);
2. Apply **u**(t) to the plant and obtain **R**(t+1);
3. Scale **R**(t+1) and apply it to the *critic*NN (both #1 and #2 are identical during this epoch), obtain λ(t+1);
4. Calculate/execute weight changes for *action*NN per Section 3; 5. If t<epoch, increment t and go to 1;
6. Go to flip epoch.

Strategy 1. Single-stage, concurrent training of *action*NN and *critic*NN.

1. Scale **R**(t) and apply it to the *action*NN, obtain **u**(t);
2. Apply **u**(t) to the plant and obtain **R**(t+1);
3. Scale **R**(t+1), apply it to the *critic*NN, obtain λ(t+1);
4. Calculate desired output λ° (t) for *critic*NN;
5. Calculate/execute weight changes for *action*NN;
6. Scale **R**(t) and apply it to *critic;*
7. Calculate/execute weight changes for *critic*NN;
8. Increment t and go to 1.

Strategy 2a. Basic flip/flop strategy (a): On-line train *critic*NN during flip epoch / on-line train *action*NN during flop epoch.

1. Scale **R**(t) and apply it to the *action*NN, obtain **u**(t);
2. Apply **u**(t) to the plant and obtain **R**(t+1);
3. Scale **R**(t+1), apply it to the *critic*NN, obtain λ(t+1);
4. Calculate desired output λ° (t) for *critic*NN;
5. Scale **R**(t) and apply it to *critic*NN;
6. Calculate/execute weight changes for *critic*NN;
7. If t<epoch, increment t and go to 1;
8. Go to flop epoch

Strategy 2b. Basic flip/flop strategy (b): Batch train *critic*NN during flip epoch / on-line train *action*NN during flop epoch.

1., 2., 3., 4. & 5. are same as in Strategy **2a.** The following are different:

6. Calculate and accumulate *critic*NN weight changes in a weight store matrix Δ**w**;
7. If t<epoch, increment t and goto 1;
8. Execute weight changes stored in Δ**w** for *critic*NN;
9. Go to flop epoch

Strategy 3a. Modified flip/flop strategy (a): On-line train *critic*NN#*1* with suspended adaptation of *critic*NN#2.

1. Scale **R**(t) and apply it to the *action*NN, obtain **u**(t);
2. Apply **u**(t) to the plant and obtain **R**(t+1);
3. Scale **R**(t+1), apply it to *critic*NN#*2*, obtain λ(t+1);
4. Calculate desired output λ° (t) for *critic*NN#*1*;
5. Scale **R**(t) and apply it to *critic*NN#*1;*
6. Calculate/execute weight changes for *critic*NN#1*;*
7. If t<epoch, increment t and go to 1;
8. Set *critic*NN#*2*=*critic*NN#*1;*
9. Go to flop epoch.

Strategy 3b. Modified flip/flop strategy (b): Batch train *critic*NN#1 with suspended adaptation of *critic*NN#2.

This entry is included for completeness. Batch training here gives same results as **2b.**

Strategy 4. Single-stage.

On-line train *action*NN and *critic*NN#1 with suspended adaptation of *critic*NN#2.

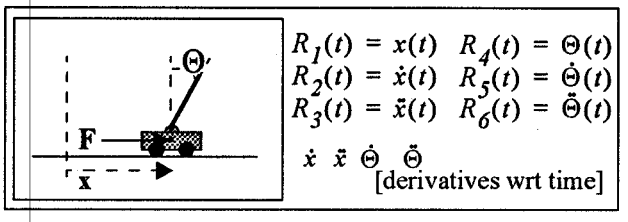
4a. Use *critic*NN#2 to update both, *action*NN and *critic*NN#1.

1. Scale **R**(t) and apply it to *action*NN, obtain **u**(t);
2. Apply **u**(t) to the plant and obtain **R**(t+1);
3. Scale **R**(t+1),apply it to *critic*NN#2, obtain λ(t+1);
4. Calculate desired output λ° (t) for *critic*NN#1;
5. Calculate/execute weight changes for *action*NN per Section 3;
6. Scale **R**(t) and apply it to *critic*NN#1;
7. Calculate/execute weight changes for *critic*NN#1;
8. If t<epoch, increment t and goto 1;
9. Set *critic*NN#2=*critic*NN#1 and go to 1.

4b. Use *critic*NN#2 to update *critic*NN#1; use *critic*NN#1 to update *action*NN.

1. Scale **R**(t) and apply it to the actionNN, obtain **u**(t);
2. Apply **u**(t) to the plant and obtain **R**(t+1);
3. Scale **R**(t+1), apply it to *critic*NN#2, obtain λ(t+1);
4. Calculate desired output λ° (t) for *critic*NN#1;
5. Scale **R**(t+1), apply it to *critic*NN#1, obtain λ(t+1);
6. Calculate/execute weight changes for *action*NN per Section 3;
7. Scale **R**(t) and apply it to *critic*NN#1;
8. Calculate/execute weight changes for *critic*NN#1;
9. If t<epoch, increment t and go to 1;
10. Set *critic*NN#2=*critic*NN#1 and go to 1.

4. Example Application



$$R_1(t) = x(t) \quad R_4(t) = \Theta(t)$$
$$R_2(t) = \dot{x}(t) \quad R_5(t) = \dot{\Theta}(t)$$
$$R_3(t) = \ddot{x}(t) \quad R_6(t) = \ddot{\Theta}(t)$$
$$\dot{x}\ \ddot{x}\ \dot{\Theta}\ \ddot{\Theta}$$
[derivatives wrt time]

Figure 2: Schematic of pole balancer. [Six dimensional state vector.]

The well known "inverted pendulum" or "pole balancer" problem, is used here as the test bed. This cart/pole system is modeled by the following equations [1]:

$$R_6(t+1) = \frac{g\sin\Theta + \cos\Theta\left[\frac{-F - ml\dot{\Theta}^2\sin\Theta + \mu_c sgn(\dot{x})}{m_c + m}\right] - \frac{\mu_p\dot{\Theta}}{ml}}{l\left[\frac{4}{3} - \frac{m(\cos\Theta)^2}{m_c + m}\right]}$$

$$R_3(t+1) = \frac{F + ml\left[\dot{\Theta}^2\sin\Theta - \ddot{\Theta}\cos\Theta\right] - \mu_c sgn(\dot{x})}{m_c + m}$$

$$R_1(t+1) = R_1(t) + \tau \bullet R_2(t);\ R_2(t+1) = R_2(t) + \tau \bullet R_3(t)$$

$$R_4(t+1) = R_4(t) + \tau \bullet R_5(t);\ R_5(t+1) = R_5(t) + \tau \bullet R_6(t)$$

τ=0.05 sec; g= 9.8 m/s^2; cart's mass m_c=1.0 kg,; pole's mass m=0.1 kg,; half pole length l=0.5m; friction coefficient of cart on track μ_c=0.0005; friction coefficient of pole on cart μ_p=0.000002; F= force applied at c.g. at time t.

We define the utility function

$$U(t) = -0.25 \bullet (\Theta(t) - \textit{desired angle})^2.$$

Balancing the pole means *desired an*gle=0.

5. Experimental Results

Each of the strategies was applied to the test-bed problem. **Strategy 2** is similar to those described in [3][2]. As will be demonstrated in the following, **modifying this strategy by suspending adaptation of the λ° process during each epoch in which the λ process is adapted (after which λ° is set equal to λ) improves the overall speed of convergence, and additionally, appears to improve the quality of the resulting *action*NN design.**

Preliminary comments:

While coding of the model equations and of the basic Backprop NN paradigm is straightforward, getting the DHP process to converge turned out being a tedious task. The available literature doesn't offer full details (page limitations!). We started with Strategy 2 (per [3]), and had to discover useful values for learning coefficients and the epoch size. Further, we discovered the efficacy of scaling the state space, and that using bias terms in the *critic*NN and particularly in the *action*NN gave problems. After this, exploring the strategies we here report followed more easily.

We simulated the inverted pendulum with the Euler iteration method, used analytical equations to compute the utility function U, and used an *action*NN and a *critic*NN. The neural network structures used were (layer format: [input/hidden/output]): ***action*NN**[6PE(lin.)/3PE(TanH)/1PE(kTanH)]; ***critic*NN**[6PE(lin.)/6PE(TanH)/6PE(lin.)]. These were trained using the basic Backpropagation algorithm. The number of PE's are specific to the plant (model).

Performance:

For the pole-balancer test bed, the training procedure was to randomly initialize all the NNs [weight range: (-.01,.01)], and to then provide a specified sequence of starting angles (with zero being the "desired" angle), allowing the system to train on each starting angle for a specified number of seconds. The measure used for comparing the various DHP strategies takes the values achieved by the primary utility function during training and accumulates these over the sequence of starting angles; the measure is here called C(j), where j labels a separate pass through the sequence of angles. In a sense, this measure incorporates the convergence speed of the

DHP strategy as well as the quality of the controller's actions along the way.

The sequence of starting angles used for the training was: (5, -10, 20, -5, -20, 10) [degrees from vertical]. The system was allowed to train on each starting angle for 30 seconds. The same sequence was run 3 times [measures C(1), C(2), C(3)], with cumulative learning. For measuring the quality of the resulting controller at the end of training, the same sequence was applied one more time, with no learning [measure C(4)].

To test generalization capability of the resulting *action*NN (controller) design, a test sequence of starting angles was presented (-23, -18, -8, 3, 13, 23) and the results measured in the same way as above [C(5)]. In this case, the measure incorporates the speed of achieving balance and quality of the controller's actions to achieve this. The results were very encouraging, so a more aggressive generalization test was performed via a second test sequence: (-38, -33, 23, 38) [C(6)], reported below. It is remarkable that in additonal tests, strategy 4a and 4b controllers successfully generalized out to 48°.

The following table shows the parameters used for each DHP strategy for the results presented. Each was selected based on experimenter's experience to yield lowest C(1). Note that the *action*NN is set up with a faster training rate than the *critic*NN. We observed that each strategy did best for a certain ratio of training rates of these two NNs.

Strategy / Train param.	1	2a	2b / 3b	3a	4a	4b
*critic*NN: learn coeff	0.03	0.03	0.02	0.2	0.15	0.2
*action*NN: learn coeff	0.1	0.2	0.5	0.6	0.6	0.5
epoch: time steps	na	3	5	3	5	5

The following table lists the value of the performance measure for each DHP train strategy, *averaged* over 4 separate training runs. C(1)=accumulated cost during first pass through train data, and C(4)=accumulated cost during pass with train data after learning is stopped. C(5)=accumulated cost during test sequence 1; C(6), test sequence 2.

	1	2a	2b /3b	3a	4a	4b
C(1)	368	740	883	209	107	160
C(4)	3.2	3.1	2.9	2.8	2.4	2.2
C(5)	5.1	7.9	7.4	7.5	6.2	5.5
C(6)	30.5	24.1	25.4	20.7	14.5	14.0

We observe in this last table that strategies 4a & 4b converge the fastest [lowest value of C(1)], and also appear to yield the best controllers [lowest values in C(4), C(5) & C(6) rows]. We note separately that strategy 1 (*when it converges*) yields controllers on par with the better ones.

The graphs shown in Figure 3 give a feel for the progress of the training process under each of the strategies. Figure 4 shows dynamic response using controller from Strategy 4b, and compared with that from 2a..

6. Conclusions

In DHP and associated methods, a "desired output" (*target*) is needed for training the *critic*NN, and this is typically calculated by running the *critic*NN one more computational cycle to provide its next-in-time output, and then use this value to compute the *target* for the present-time cycle. The error term is calculated and the *critic*NN update is performed in the usual way. These approaches typically use a "copy" of the *critic*NN to perform (or at least explain) the calculation of the *target*, and both copies are updated at the same time. Since the *critic*NN that calculates the *target* is changing with each update, it provides a "moving target" for the *critic*NN training. In this paper, we introduce a **real** copy of the *critic*NN (*critic*NN#2) for making the *target* calculations, and very importantly, this ***critic*NN#2 is trained differently than is *critic*NN#1**. The idea is to provide the *target* from a stable platform during an epoch while adapting *critic*NN#1. Then at the end of the epoch, *critic*NN#2 is made identical to the then-current adapted state of *critic*NN#1, and a new epoch starts. In this way, both the *critic*NN#1 and the *action*NN can be trained on-line during each epoch, with a faster overall convergence than the older approach. Further, the lower relative values of C(4), C(5) and C(6) suggest a "better" design is developed for the controller (*action*NN).

References

[1] Barto, A., Sutton, R. & Anderson, C. " Neuronlike Adaptive Elements that can Solve Difficult Learning Control Problems" in *IEEE Transactions on Systems, Man and Cybernetics*, Vol. SMC-13, No.5, Sep/Oct 1983.

[2] Prokhorov, D. and Wunsch, D. "Advanced Adaptive Critic Designs", *PROCEEDINGS WCNN'96*, pp. 83-87, San Diego, Erlbaum, Sept. 1996.

[3] Santiago, R., presentation at the First Joint Mexico-US International Workshop on Neural Networks and Neurocontrol, Playacar, Mexico, Sept. 1995.

[4] Santiago, R. and Werbos, P. "New Progress Towards Truly Brain-Like Intelligent Control", *PROCEEDINGS WCNN '94*, pp. I-2toI-33, San Diego, Erlbaum, 1994.

[5] Visnevski, N. and Propkhorov, 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.

[6] 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, D.A. and Sofge, D.A., eds.), Van Nostrand Reinhold, New York, NY, 1992

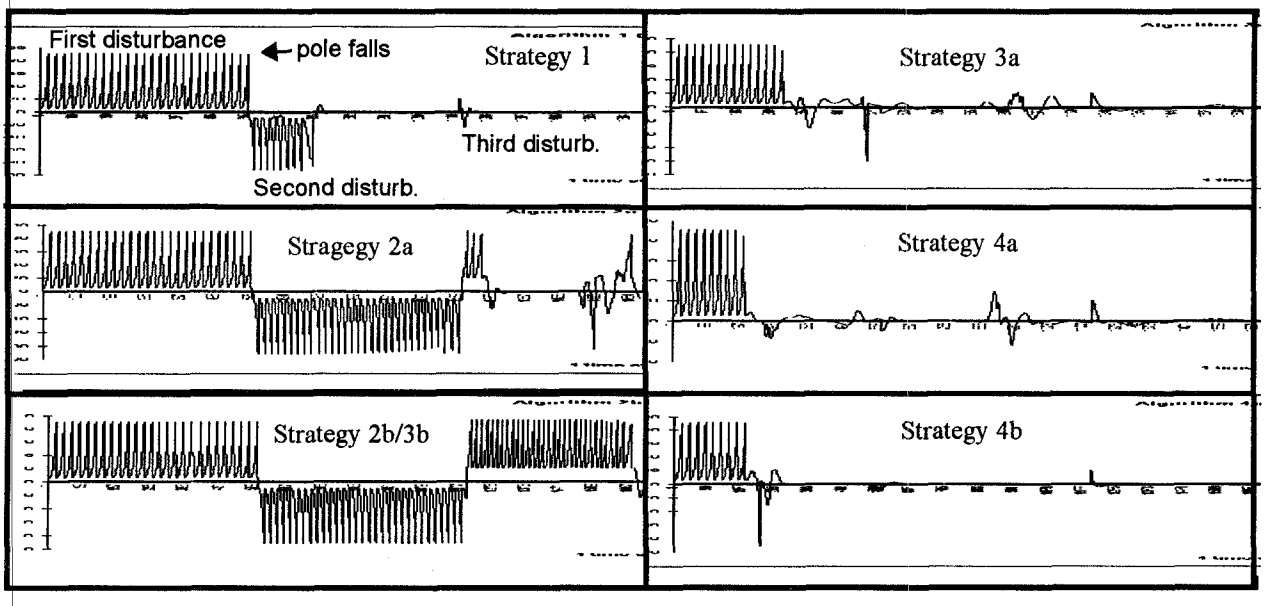


Figure 3: Progress of training process under each of the strategies.

[Pole angles during the first 80 sec. of training (~3 disturbances). Note how fast Strategies 4a & 4b learn.]

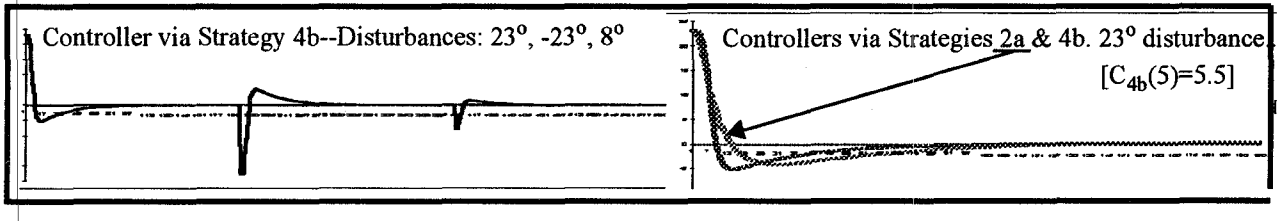


Figure 4: Pole angles during part of test sequence.

[Markings below and parallel to axis are plotting artifacts.]