Action-Dependent Adaptive Critic Designs*

DERONG LIU, XIAOXU XIONG, and YI ZHANG
University of Illinois
Department of Electrical Engineering and Computer Science
851 S. Morgan Street, Chicago, IL 60607, U. S. A.
Email: {liu, yzhang2, xxiong}@eecs.uic.edu

Abstract

In this work, we study a class of action-dependent adaptive critic designs. Conventional adaptive critic designs contain three basic modules: Critic, model, and action. Each of the three modules can be implemented using a neural network. By combining the critic network and the model network to form a new critic network, we propose a form of action-dependent adaptive critic designs where the critic network implicitly includes a model network in it. An important feature of the present design is that the proposed action-dependent adaptive critic designs can be applied to on-line learning control applications. We also provide details about the training of the neural networks used in the present design. The present training approach makes it possible the use of many readily available neural network training algorithms and tools without modifications. We employ the pole balancing problem in our simulation study to show the applicability of the present results.

1 Introduction

Suppose that one is given a discrete-time nonlinear (time-varying) system

$$x(t+1) = F[x(t), u(t), t] \tag{1}$$

where $x \in R^n$ represents the state vector of the system and $u \in R^m$ denotes the control action. Suppose that one associates with this system the performance index (or cost)

$$J[x(i), i] = \sum_{k=i}^{\infty} \gamma^{k-i} U[x(k), u(k), k] \tag{2}$$

where U is called the utility function and γ is the discount factor with $0 < \gamma \leq 1$. Note that J is dependent on the initial time i and the initial state $x(i)$, and it is referred to as the cost-to-go of state $x(i)$. The objective is to choose the control sequence $u(k)$, $k = i, i+1, \cdots$, so that the J function (i.e., the cost) in (2) is *minimized*. Dynamic programming is based on Bellman's *principle of optimality* [5], [8]: An optimal (control) policy has the property that no matter what previous decisions have been, the remaining decisions must constitute an optimal policy with regard to the state resulting from those previous decisions.

Suppose that one has computed the optimal cost $J^*[x(t+1), t+1]$ from time $t+1$ on for all possible states $x(t+1)$, and that one has also found the optimal control sequences from time $t+1$ on. The optimal cost results when the optimal control sequence $u^*(t+1)$, $u^*(t+2)$, $\cdots$, is applied to the system with initial state $x(t+1)$. Note that the optimal control sequence depends on $x(t+1)$. If one applies an arbitrary control $u(t)$ at time t and then uses the known optimal control sequence from $t+1$ on, the resulting cost will be $U[x(t), u(t), t] + \gamma J^*[x(t+1), t+1]$, where $x(t)$ is the state at time t and $x(t+1)$ is determined by (1). According to Bellman, the optimal cost from time t on is equal to

$$J^*[x(t), t] = \min_{u(t)} \Big(U[x(t), u(t), t] + \gamma J^*[x(t+1), t+1] \Big).$$

The optimal control $u^*(t)$ at time t is the $u(t)$ that achieves this minimum, i.e.,

$$u^*(t) = \arg \min_{u(t)} \Big(U[x(t), u(t), t] + \gamma J^*[x(t+1), t+1] \Big). \tag{3}$$

Equation (3) is the principle of optimality for discrete-time systems. Its importance lies in the fact that it allows one to optimize over only one control vector at a time by working *backward* in time. In other words, any strategy of action that minimizes J in the short term will also minimize the sum of U over all future times.

*This work was supported by the National Science Foundation under Grant ECS-9996428

0-7803-7044-9/01/$10.00 ©2001 IEEE

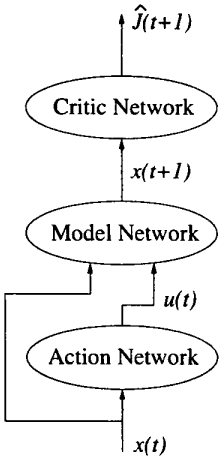

Figure 1: The three modules in a typical adaptive critic design.

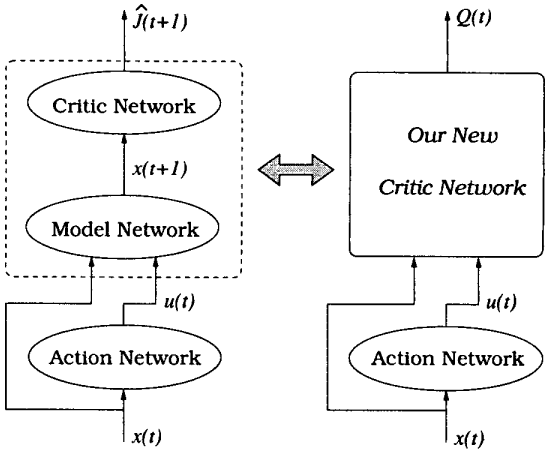

Figure 2: A new critic network.

2 Adaptive Critic Designs

Adaptive critic designs (ACDs) have received increasing attention recently (cf. [3], [6], [12], [13], [16]–[20]). ACD is defined as a scheme that *approximates dynamic programming in the general case*, i.e., approximates optimal control over time in noisy, nonlinear environments. There are many problems in practice which can be formulated as cost maximization or minimization problems. Dynamic programming is a very useful tool in solving these problems. However, it is often computationally untenable to run true dynamic programming due to the backward numerical process required for its solutions, i.e., as a result of the "curse of dimensionality" [5]. Over the years, progress has been made to circumvent the "curse of dimensionality" by building a system, called "critic," to approximate the cost function in dynamic programming (cf. [17], [20]). The idea is to approximate dynamic programming solutions by using a function approximation structure such as neural networks to approximate the cost function.

A typical design of ACD consists of three modules–Critic, Model, and Action [13], [17], [20], as shown in Figure 1. In this case, the critic network outputs an estimate of the J function in equation (2). This is done by minimizing the following error measure over time,

$$\begin{aligned} \|E_h\| &= \sum_t E_h(t) \\ &= \sum_t \left[\hat{J}(t) - U(t) - \gamma \hat{J}(t+1)\right]^2 \end{aligned} \tag{4}$$

where $\hat{J}(t) = \hat{J}[x(t), u(t), t, W_C]$ and W_C represents the parameters of the critic network. The function U is the same utility function as the one in (2) which indicates the performance of the overall system (see examples in [3], [12], [13], [17], [19]). When $E_h(t) = 0$ for all t, (4) implies that

$$\begin{aligned} \hat{J}(t) &= U(t) + \gamma \hat{J}(t+1) \\ &= U(t) + \gamma[U(t+1) + \gamma \hat{J}(t+2)] \\ &= \cdots \\ &= \sum_{k=t}^{\infty} \gamma^{k-t} U(k) \end{aligned} \tag{5}$$

which is exactly the same as the cost in (2). It is therefore clear that by minimizing the error function in (4), we will have a neural network trained so that its output becomes an estimate of the cost function defined in (2). The model network in an ACD predicts $x(t+1)$ given $x(t)$ and $u(t)$, i.e., it learns the mapping given in (1). The model network can be trained previously off-line [13], [16], [17], [19], or trained in parallel with the critic and action networks [14]. The action network is trained with the objective of minimizing $\hat{J}(t+1)$, through the use of the action signal $u(t) = u[x(t), t, W_A]$. Once an action network is trained this way, i.e., trained by minimizing the output of critic network, the action network will generate a control action signal which is the optimal control action or is very close to the optimal control action (depending on how well the critic network is trained). Recall that the goal of dynamic programming is to obtain an optimal control sequence as in (3), which will minimize the J function in (2). During the training of the action network, the three networks will be connected as shown in Figure 1.

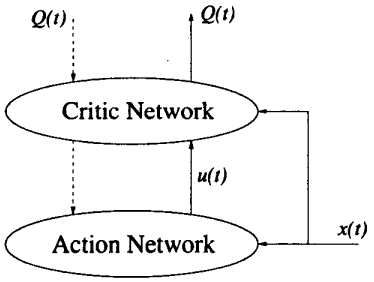

Figure 3: A typical scheme of an action-dependent adaptive critic design.

3 Action-Dependent Adaptive Critic Designs

In this section, we propose a modified version of the ACD introduced in the preceding section. We will consider a new critic network as defined in Figure 2. We can see that the new critic network will include the explicit model network in Figure 1 as part of its internal state. This gives us a few advantages including the simplification of the overall system design and the feasibility of the present approach to applications where a model network may be very difficult to obtain.

In the following we will describe in detail how the action network and the new critic network (therefore, critic network in the sequel) in Figure 2 are trained. We will use Figure 3 to refer to our new ACD. The new design is actually an ACD with an embedded model network. Since the ACD in Figure 3 includes the control action signal as input to the critic network, it is therefore a *model-free action-dependent ACD* as defined in the literature [12], [13], [17], [20]. Note that in the literature, action-dependent ACDs include both model-free and model-based versions.

Consider the ACD shown in Figure 3. The critic network in this case will be trained by minimizing the following error measure over time,

$$\begin{aligned} \|E_q\| &= \sum_t E_q(t) \\ &= \sum_t [Q(t-1) - U(t) - \gamma Q(t)]^2 \end{aligned} \tag{6}$$

where $Q(t) = Q[x(t), u(t), t, W_C]$. When $E_q(t) = 0$ for all t, (6) implies that

$$\begin{aligned} Q(t) &= U(t+1) + \gamma Q(t+1) \\ &= U(t+1) + \gamma[U(t+2) + \gamma Q(t+2)] \\ &= \cdots \\ &= \sum_{k=t+1}^{\infty} \gamma^{k-t-1} U(k). \end{aligned} \tag{7}$$

Clearly, comparing (2) and (7), we have now $Q(t) = J[x(t+1), t+1]$. Therefore, when minimizing the error function in (6), we have a neural network trained so that its output becomes an estimate of the cost function defined in (2) for $i = t+1$, i.e., the value of the cost function in the immediate future.

The training samples for the critic network are obtained over a trajectory starting from $x(0)$ at $t = 0$. Starting from $x(0)$, we can apply $u(0)$ to equation (1) (or the system to be controlled for on-line applications) to obtain $x(1)$, and then apply $x(1)$ and $u(1)$ to equation (1) to obtain $x(2)$, and so on. The action signal $u(t), t = 0, 1, \cdots$, will be generated from an initial action network which is initialized with random weights. The trajectory can be either over a fixed number of time steps (e.g., 300 consecutive points in [13]) or from $t = 0$ until the final state is reached (e.g., the plane is crashed or landed in the autolanding problem [12]).

The input-output relationship of the critic network in Figure 3 is given by

$$Q(t) = Q\left[x(t), u(t), t, W_C^{(p)}\right]$$

where $W_C^{(p)}$ represents the weights of the critic network after the pth weight update. There are two approaches to train the critic network according to (6) in the present case which are described next.

(1) *Backward-in-time:* We can train the critic network at time t, with the output target given by $[Q(t-1) - U(t)]/\gamma$. The training of the critic network is to realize the mapping given by

$$C: \begin{Bmatrix} x(t) \\ u(t) \end{Bmatrix} \rightarrow \left\{ \frac{1}{\gamma}[Q(t-1) - U(t)] \right\}. \tag{8}$$

In this case, we consider $Q(t)$ in (6) as the output from the network to be trained and the target output value is calculated using the critic network output at time $t-1$.

(2) *Forward-in-time:* We can train the critic network at time $t-1$, with the output target given by $U(t) + \gamma Q(t)$. The training of the critic network is to realize the mapping given by

$$C: \begin{Bmatrix} x(t-1) \\ u(t-1) \end{Bmatrix} \rightarrow \{U(t) + \gamma Q(t)\}. \tag{9}$$

In this case, we consider $Q(t-1)$ in (6) as the output from the network to be trained and the target output value is calculated using the critic network output at time t.

The two approaches described here are based on two different view points of equation (6). However, the backward-in-time approach may have problems in numerical implementations. For example, when $0 < \gamma < 1$, the target value in (8) will tend to increase in magnitude [unless if we choose $U(t) = 0$ for most t]. Our experiments have shown that if $U(t)$ is chosen as an error function (which will be non-zero until the control objective is reached), the training of the critic network using backward-in-time approach will be numerically unstable due to the increase (without bound) of the target value in (8). In the following we will concern ourselves with the forward-in-time approach.

After the critic network's training is finished, the action network's training starts with the objective of minimizing $Q(t)$. For the training of the action network, we start with $x(0)$ and the action network gives $u(0) = u[x(0), W_A^{(p)}]$. We then use equation (1) (or from the system to be controlled for on-line applications) to obtain $x(1)$. Using the action network, $x(1)$ gives $u(1) = u[x(1), W_A^{(p)}]$ and $x(2)$ is obtained from the plant at the next time step [or from the simulation of equation (1)]. This process continues until all the necessary training patterns are collected. The goal of the action network training is to minimize the output of the critic network $Q(t)$. In this case, we can choose the target of the action network training as zero or negative values, i.e., we will train the action network so that the output of the critic network becomes as small as possible. The desired mapping which will be used for the training of the action network in Figure 3 is given by

$$A: \{x(t)\} \rightarrow \{0(t)\} \tag{10}$$

where $0(t)$ indicates the target of zero or negative values. We note that during the training of the action network, it will be connected to the critic network as shown in Figure 3, and the target in (10) is for the output of the critic network.

After the action network's training cycle is completed, one may check the system's performance, then stop or continue the training procedure by going back to the critic network's training cycle again, if the performance is not acceptable yet.

Remark 3.1 In the preceding, we did not provide any details about the weight updating algorithm. In many existing works on ACDs [13], [16], [17], [20], weight updating algorithms have been proposed based on primarily gradient based approach. When deriving the gradient based training algorithms, care must be taken regarding the derivative of $\hat{J}(t+1)$ or $Q(t)$ with respect to W_C (cf. [19]). In the present approach, we concentrate on training neural networks with clearly identified training patterns for each training step. Once the training patterns are identified and collected, we can use readily available algorithms in the literature to train our (critic and action) networks. We will test the Levenberg-Marquardt algorithm [9] and the simple gradient method [10] in our experiments which are implemented in MATLAB Neural Network Toolbox [7] as `trainlm` function and `traingd` function, respectively. ∎

4 The Pole Balancing Problem

We consider the pole balancing (inverted pendulum) problem [2] in our simulation study to evaluate the ACD presented in the preceding section. This is the problem of learning to balance an upright pole, whose bottom is attached by a pivot to a cart that travels along a track. The state of this system is given by the pole's angle and angular velocity and the cart's horizontal position and velocity. The only available control actions are to exert forces of fixed magnitude on the cart that push it to the left or right.

The event of the pole falling past a certain angle or the cart running into the bounds of its track is called a *failure*. A sequence of forces must be applied so that failures can be avoided by balancing the pole in the center of the track. A naive controller, before learning much about the task, will be unable to avoid failures. The pole and cart system is reset to its initial state after each failure and the controller must learn to balance the pole for as long as possible.

The cart-pole system is described by

$$\ddot{\theta}(t) = \frac{mg \sin\theta(t) - \cos\theta(t)\left(f(t) + m_p l \dot{\theta}^2(t) \sin\theta(t)\right)}{(4/3)ml - m_p l \cos^2\theta(t)}$$

and

$$\ddot{x}(t) = \frac{f(t) + m_p l\left(\dot{\theta}^2(t)\sin\theta(t) - \ddot{\theta}(t)\cos\theta(t)\right)}{m}.$$

The parameters used in the cart-pole system are: $g = 9.8$ m/s^2, $m = 1.1$ kg, $m_p = 0.1$ kg, $l = 0.5$ m. The step size suggested in [2] for numerical integration is 0.02 seconds. The constraints of the system are given by $-12° < \theta < 12°$ and $-2.4 < x < 2.4$ m. These constraints indicates that a failure occurs when either $|\theta| \geq 12°$ or $|x| \geq 2.4$. The control force has a fixed magnitude of 10, i.e., $f = -10$ N or 10 N.

The U function: The control objective of this problem is to avoid failures. This objective can be indicated by the following U function chosen similarly as given in [2],

$$U(t) = \begin{cases} 1, & \text{if } |\theta(t)| \geq 12^\circ \text{ or } |x(t)| \geq 2.4 \text{ m} \\ 0, & \text{otherwise.} \end{cases} \quad (11)$$

When we minimize the sum of the U function over the period T of a trial given by $\sum_{k=0}^{T} \gamma^k U(k)$, we will keep the cart-pole system from failure for as long as possible.

The critic network: The critic network is chosen as a 5–6–1 structure with 5 input neurons and 6 hidden layer neurons. The 5 inputs are the 4 states $\theta(t)$, $\dot{\theta}(t)$, $x(t)$, and $\dot{x}(t)$, and the output of the action network $u(t)$. For the critic network, the hidden layer uses the sigmoidal function given by

$$y = \frac{1 - e^{-x}}{1 + e^{-x}},$$

i.e., the `tansig` function in MATLAB [7], and the output layer uses the linear function `purelin`. Utilizing the MATLAB Neural Network Toolbox, we have applied `traingd` (simple gradient descent) and `trainlm` (the Levenberg-Marquardt algorithm) for the training of the critic network. We note that other algorithms implemented in MATLAB, such as `traingda`, `traingdm`, `traingdx` are also equally applicable. We use the forward-in-time approach outlined earlier (though the backward-in-time approach works as well for the present example due to the way the U function is defined). We employ sequential training for the critic network, i.e., the training is performed at each time step using the immediately obtained training sample. We use $\gamma = 0.9$ in the present experiments.

The action network: The structure of the action network is chosen as 4–6–1 with 4 input neurons and 6 hidden layer neurons. The 4 inputs are $\theta(t)$, $\dot{\theta}(t)$, $x(t)$, and $\dot{x}(t)$. Both the hidden layer and the output layer use the sigmoidal function `tansig`. The training algorithms we choose to use are `traingd` and `trainlm`. We employ batch training for the action network, i.e., the network is trained after each trial. The output of the action network is $u(t)$ which is constrained to $(-1, 1)$. The control action applied to the cart is computed as $f(t) = 10\,\text{sign}\,[u(t)]$.

Comments on existing works: The pole balancing problem has been considered by many researchers as a good benchmark for testing new control and learning algorithms. In particular, this problem is considered in [1], [4], [11], and [15] as a benchmark for testing ACDs/reinforcement learning. In the implementations of [1] and [4], the U function is chosen as the same as in the present study and the action signals are generated using a random process biased by the learning process. In the implementations of [11] and [15] for testing ACDs, the U function is chosen as a function associated with the system states, which implies that a state measurement mechanism must be employed. We note that the implementations of [11] and [15] are *not* based on action-dependent ACDs.

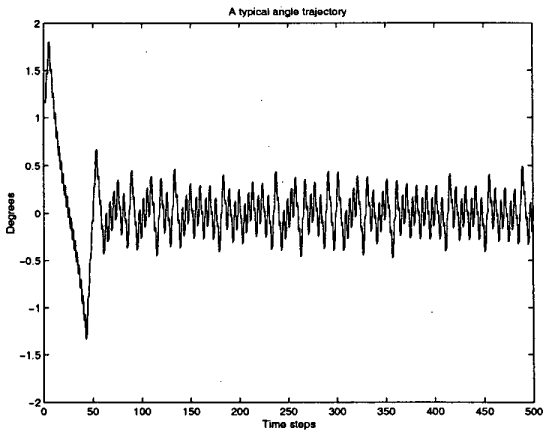

Figure 4: The angle trajectory from a successful trial.

Results: We display in Figure 4 a typical trajectory of the pole angle from a successful trial using the present action-dependent ACD. A success in our study is defined as balancing the pole for at least 30 minutes (90000 steps) [2], [15], even though the display in Figure 4 is only for 500 steps. We conduct our experiments starting from randomly initialized critic and action networks. We choose randomly initial θ in the range of $(-12^\circ, 12^\circ)$ and zero values for the other three states for each trial in our experiments until a trial which successfully balances the pole for at least 90000 steps. We repeat the experiments for a total of 100 times. The average number of trials which leads to the first successful balancing among the 100 experiments is 26. We note that in the study of [1], the authors do not reset each trial with random initial states because they use a biased random process for generating action signals. In our implementation, however, it seems necessary that we reset each trial with random initial states (e.g., random initial θ). If we reset each trial with the same or zero initial states, the learning process will often get stuck after a few trials leading to no further progress in the learning process.

5 Concluding Remarks

Starting from the conventional model-based adaptive critic designs, we argued that by combining the model network and the critic network, we can obtain an equivalent form of model-free action-dependent adaptive critic designs. An important feature of the present design is that the proposed designs can be applied to on-line learning control applications. We tested the present approach using the pole balancing problem and our simulation showed very good performance results. We emphasize that in our simulation studies, we used the commercially available MATLAB package for the training of our networks.

References

[1] C. W. Anderson, "Learning to control an inverted pendulum using neural networks," *IEEE Contr. Syst. Magazine*, vol. 9, pp. 31–37, Apr. 1989.

[2] C. W. Anderson and W. T. Miller III, "Challenging control problems," in *Neural Networks for Control* (Appendix A), Edited by W. T. Miller III, R. S. Sutton, and P. J. Werbos, Cambridge, MA: The MIT Press, 1990.

[3] S. N. Balakrishnan and V. Biega, "Adaptive-critic-based neural networks for aircraft optimal control," *J. Guidance, Contr., Dynamics*, vol. 19, pp. 893–898, July 1996.

[4] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-13, pp. 834–846, Sept./Oct. 1983.

[5] R. E. Bellman, *Dynamic Programming*, Princeton, NJ: Princeton University Press, 1957.

[6] D. P. Bertsekas and J. N. Tsitsiklis, *Neuro-Dynamic Programming*, Belmont, MA: Athena Scientific, 1996.

[7] H. Demuth and M. Beale, *Neural Network Toolbox User's Guide*, Natick, MA: MathWorks, 1998 (Version 3).

[8] S. E. Dreyfus and A. M. Law, *The Art and Theory of Dynamic Programming*, New York, NY: Academic Press, 1977.

[9] M. T. Hagan and M. B. Menhaj, "Training feedforward networks with the Marquardt algorithm," *IEEE Trans. Neural Networks*, vol. 5, pp. 989–993, Nov. 1994.

[10] S. Haykin, *Neural Networks: A Comprehensive Foundation*, Upper Saddle River, NJ: Prentice Hall, 1999.

[11] G. G. Lendaris and C. Paintz, "Training strategies for critic and action neural networks in dual heuristic programming method," *Proc. 1997 Int. Conf. Neural Networks*, Houston, TX, June 1997, pp. 712–717.

[12] D. V. Prokhorov, R. A. Santiago, and D. C. Wunsch, "Adaptive critic designs: A case study for neurocontrol," *Neural Networks*, vol. 8, pp. 1367–1372, 1995.

[13] D. V. Prokhorov and D. C. Wunsch, "Adaptive critic designs," *IEEE Trans. Neural Networks*, vol. 8, pp. 997–1007, Sept. 1997.

[14] R. E. Saeks, C. J. Cox, K. Mathia, and A. J. Maren, "Asymptotic dynamic programming: Preliminary concepts and results," *Proc. 1997 Int. Conf. Neural Networks*, Houston, TX, June 1997, pp. 2273–2278.

[15] R. A. Santiago and P. J. Werbos, "New progress towards truly brain-like intelligent control," *Proc. 1994 World Cong. Neural Networks*, San Diego, CA, June 1994, pp. 27–33.

[16] P. J. Werbos, "Neurocontrol and supervised learning: An overview and evaluation," in *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches* (Chapter 3), Edited by D. A. White and D. A. Sofge, New York, NY: Van Nostrand Reinhold, 1992.

[17] P. J. Werbos, "Approximate dynamic programming for real-time control and neural modeling," in *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches* (Chapter 13), Edited by D. A. White and D. A. Sofge, New York, NY: Van Nostrand Reinhold, 1992.

[18] P. J. Werbos, "Building and understanding adaptive systems: A statistical/numerical approach to factory automation and brain research," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-17, pp. 7–20, Jan./Feb. 1987.

[19] P. J. Werbos, "Consistency of HDP applied to a simple reinforcement learning problem," *Neural Networks*, vol. 3, pp. 179–189, 1990.

[20] P. J. Werbos, "A menu of designs for reinforcement learning over time," in *Neural Networks for Control* (Chapter 3), Edited by W. T. Miller III, R. S. Sutton, and P. J. Werbos, Cambridge, MA: The MIT Press, 1990.