October 1995: Accepted for publication in *Adaptive Behavior*

Reinforcement Learning Applied to a Differential Game

Mance E. Harmon*
Wright Laboratory

Leemon C. Baird III**
United States Air Force Academy

A. Harry Klopf
Wright Laboratory

An application of reinforcement learning to a linear-quadratic, differential game is presented. The reinforcement learning system uses a recently developed algorithm, the residual-gradient form of advantage updating. The game is a Markov decision process with continuous time, states, and actions, linear dynamics, and a quadratic cost function. The game consists of two players, a missile and a plane; the missile pursues the plane and the plane evades the missile. Although a missile and plane scenario was the chosen test-bed, the reinforcement learning approach presented here is equally applicable to biologically based systems, such as a predator pursuing prey. The reinforcement learning algorithm for optimal control is modified for differential games to find the minimax point rather than the maximum. Simulation results are compared to the analytical solution, demonstrating that the simulated reinforcement learning system converges to the optimal answer. The performance of both the residual-gradient and non-residual-gradient forms of advantage updating and Q-learning are compared, demonstrating that advantage updating converges faster than Q-learning in all simulations. Advantage updating is also demonstrated to converge regardless of the time step duration; Q-learning is unable to converge as the time step duration grows small.

Key words: reinforcement learning; advantage updating; dynamic programming; differential games

* WL/AAAT Bldg. 635, 2185 Avionics Circle, Wright-Patterson Air Force Base, OH 45433-7301; harmonme@aa.wpafb.mil

** Current address: HQ USAFA/DFCS, 2354 Fairchild Dr. Suite 6K41, USAFA, CO 80840-6234; baird@cs.usafa.af.mil

Incremental value iteration (Bertsekas, 1987) is a form of dynamic programming that has been used to solve reinforcement learning problems. However, value iteration is often impractical for reinforcement learning because it requires that a model be known or learned. Furthermore, for a continuum of states and actions, it requires the calculation of the maximum of an infinite set of integrals to perform one update.

Watkins (1989) proposed an algorithm called *Q*-learning that avoids the problems of value iteration. It does not need a model of the system and is guaranteed to converge, at least for the discrete case with a finite number of states and actions. However, the number of updates required for training scales poorly with the duration of the time step. As the time step grows shorter, *Q*-learning learns more slowly and, in the limit of continuous time, it is impossible for *Q*-learning to learn the correct policy. This problem led to the development of *advantage updating* by Baird (1993). Advantage updating is direct, is guaranteed to converge, and is appropriate for continuous-time systems or systems with small time steps.

Another area of study in reinforcement learning has been the use of function-approximation systems for the storage of the information used with a reinforcement learning algorithm. Proofs of convergence that are valid when using a look-up table are not applicable when the same information is stored in a function approximator. In many cases the parameters of the function approximator may oscillate or grow to infinity. Baird and Harmon (1994) proposed the class of residual-gradient algorithms that guarantees convergence when using an arbitrary function approximator for the storage of the information in a reinforcement learning system. In this article, we evaluate advantage updating and residual gradient algorithms in comparative assessments with *Q*-learning, using the problem of missile avoidance as a test bed.

Four reinforcement learning algorithms are applied to a linear-quadratic, differential game. The results of computer simulations are presented. Section 1 presents a brief discussion of Markov decision processes, along with definitions of reinforcement learning, expected total discounted reinforcement, and the discount factor. Section 2 describes residual-gradient algorithms (Baird & Harmon, In preparation) and discusses the difficulty of combining reinforcement learning algorithms with function approximation systems. Section 3 describes *Q*-learning (Watkins, 1989) and Section 4 describes advantage updating (Baird, 1993). Section 5 discusses differential games (Isaacs, 1965) and explains minimax, maximin, and the saddle point. Section 6 describes the computer simulation that constituted the test bed and gives the derivation of

advantage updating for differential games. Section 7 presents experimental results for a reinforcement learning system employing the residual-gradient form of advantage updating in learning a differential game. Also given are comparative results of advantage updating and *Q*-learning, in both the original and residual-gradient forms. Section 8 discusses these experimental results and outlines possible future research directions.

1 REINFORCEMENT LEARNING SYSTEMS

Typically, a reinforcement learning system uses a set of real-valued parameters to store the information that is learned. When a parameter is updated during learning, the notation:

$$W \leftarrow K \tag{1}$$

represents the operation of instantaneously changing the parameter W so that its new value is K, whereas:

$$W \xleftarrow{\alpha} K \tag{2}$$

represents the operation of moving the value of W toward K. This is equivalent to:

$$W_{new} \leftarrow (1-\alpha)W_{old} + \alpha K \tag{3}$$

where the learning rate α is a small positive number. Appendix A summarizes these and other notational conventions.

1.1 MARKOV SEQUENTIAL DECISION PROCESSES

A Markov sequential decision process (MDP) is a system that changes its state based upon its current state and an action chosen by a controller. The set of possible states and the set of possible actions may each be finite or infinite. At time t, the controller chooses an action, u_t, based upon the state of the MDP, x_t. The MDP then transitions to a new state, $x_{t+\Delta t}$, where Δt is the duration of a time step. The state transition may be stochastic, but the probability, $P(u_t, x_t, x_{t+\Delta t})$, of transitioning from state x_t to state $x_{t+\Delta t}$ after performing action u_t is a function of only x_t, $x_{t+\Delta t}$, and u_t, and is not affected by previous states or actions. If there are a finite number of possible states and actions, then $P(u_t, x_t, x_{t+\Delta t})$ is a probability. If there are a continuum of possible states or actions, then $P(u_t, x_t, x_{t+\Delta t})$ is a probability density function (PDF). If time is continuous rather than discrete, then $P(u_t, x_t, \dot{x}_t)$ is the probability that action u_t

will cause the velocity to be $\dot{x}_t$. The MDP also sends the controller a scalar value known as reinforcement. If time is discrete, then the total reinforcement received by the controller during time step t is $R_{\Delta t}(x_t,u_t)$. If time is continuous, then the rate of reinforcement at time t is $r(x_t,u_t)$.

1.2 REINFORCEMENT LEARNING

The most commonly defined reinforcement learning problem is that of finding actions that maximize the *expected total discounted reinforcement* which, for continuous time, is defined as:

$$\left\langle \int_0^\infty \gamma^t r(x_t, u_t) dt \right\rangle \tag{4}$$

where $\langle \bullet \rangle$ denotes expected value and $0<\gamma\leq 1$ is the *discount factor* which determines the relative significance of earlier versus later reinforcement. For discrete time, the total discounted reinforcement received during one time step of duration Δt when performing action u_t in state x_t is defined as:

$$R_{\Delta t}(x_t, u_t) = \int_t^{t+\Delta t} \gamma^{\tau - t} r(x_\tau, u_\tau) d\tau \tag{5}$$

The goal for a discrete-time controller is to find actions that maximize the expected total discounted reinforcement:

$$\left\langle \sum_{i=0}^{\infty} \left(\gamma^{\Delta t}\right)^i R_{\Delta t}(x_{i\cdot\Delta t}, u_{i\cdot\Delta t}) \right\rangle \tag{6}$$

This expression is often written with Δt not shown and with γ chosen to implicitly reflect Δt, but is written here with the Δt shown explicitly so that expression (6) will reduce to expression (4) in the limit as Δt goes to zero. A *policy*, $\pi(x)$, is a function that specifies a particular action for the controller to perform in each state x. The *optimal policy* for a given MDP, $\pi^*(x)$, is a policy such that choosing $u_t=\pi^*(x_t)$ results in maximizing the total discounted reinforcement for any choice of starting state. If reinforcement is bounded, then at least one optimal policy is guaranteed to exist. The *value* of a state, $V^*(x)$, is the expected total discounted reinforcement received when starting in state x and choosing all actions according to an optimal policy. The functions stored in a learning system at a given time are represented by variables without superscripts such as π, V, A, or Q. The

optimal functions that are being approximated are represented by * superscripts, such as π^*, V^*, A^*, or Q^*.

2 RESIDUAL-GRADIENT ALGORITHMS

Reinforcement learning algorithms are sometimes based on incremental dynamic programming. Dynamic programming algorithms can be guaranteed to converge to optimality when used with look-up tables, yet be completely unstable when combined with function-approximation systems (Baird , 1995a). It is possible to derive an algorithm that has guaranteed convergence for continuous time problems for a particular function approximation system (Bradtke, 1993), but it would be useful to derive an algorithm that works with any function approximation system. One solution to this problem is to modify the learning algorithm. The goal of a typical reinforcement learning system applied to a deterministic MDP is to find a function $V(x)$ that satisfies the following equation for all x:

$$V(x_t) = \max_u \left(R_{t+\Delta t} + \gamma V(x_{t+\Delta t}) \right) \tag{7}$$

The unique solution to equation (7) is the optimal value function $V^*(x)$. Equation (7) plays a role similar to the Bellman equation in dynamic programming (Bertsekas, 1987); it is actually a special case of the Bellman equation for deterministic MDPs. A function approximation system can be used to represent the value function $V(x)$ during learning. Tesauro successfully applied this idea to Q-learning for playing backgammon (Tesauro, 1990 & 1992). Before learning, the initial value function may not satisfy equation (7), so it is useful to define the *Bellman residual error*, E, as used in Williams and Baird (1993) and defined here as equation (8):

$$E(x_t) = \max_u \left(R_{t+1} + \gamma V(x_{t+1}) \right) - V(x_t) \tag{8}$$

For a nondeterministic MDP, the Bellman residual would be the expected value of the right side of equation (8). However, the derivation presented here assumes a deterministic MDP. The Bellman residual is zero for all states if and only if the function V satisfies the Bellman equation. Any reinforcement learning algorithm based on dynamic programming will have one or more equations corresponding to the Bellman equation, and will therefore have one or more Bellman residuals that should go to zero. A *residual-gradient* algorithm is a reinforcement learning algorithm that performs gradient descent on the mean squared Bellman residual. If the Bellman residual goes to

zero, this generates a function satisfying the equations corresponding to the Bellman equation.

When a backpropagation network (Rumelhart, Hinton, & Williams, 1986) is trained to learn a single, differentiable function, it is a true gradient-descent algorithm, so convergence is guaranteed in the following sense. If backpropagation is used to learn from a finite training set, and if the weights are changed only after each full pass through the training set, then this is called *epoch-wise* training. When backpropagation is used in an epoch-wise training mode, it calculates the exact gradient of the mean squared error, and changes the weights proportionally. For a sufficiently small, constant learning rate, and a smooth error function, backpropagation will converge to within an arbitrarily small distance of a local minimum of mean squared error. If there are an infinite number of training examples that are drawn at random according to some probability distribution, and if the weights are changed after each training example instead of after each epoch, then the weights follow an approximate gradient, but will still converge to a local minimum of mean squared error if the learning rate decreases over time at an appropriate rate. These convergence properties are not unique to the backpropagation algorithm; they are true of any algorithm that performs gradient descent on a differentiable, nonnegative, mean error function.

Pure backpropagation is guaranteed to converge when trained to learn a single function. Backpropagation is not guaranteed to converge when combined with a dynamic-programming-based algorithm, such as *Q*-learning. Combining backpropagation with *Q*-learning results in an algorithm that may not converge because the "desired" output of the network for a given input keeps changing. The network is performing gradient descent on an error function that keeps changing. The network is trying to follow a moving target; it is no longer following the gradient of a single, unchanging error function. This problem is illustrated in Appendix B, where the weights almost never converge, even though the algorithm is performing epoch-wise training over a finite set of training examples. Bradtke (1993) has developed algorithms that do not require lookup tables or linearly independent states to guarantee convergence. Although these algorithms are guaranteed to converge when learning to control a linear quadratic regulator (LQR) problem with a quadratic function-approximation system, we will present a class of algorithms that guarantee convergence when learning to control nonlinear systems with arbitrary function approximators.

Residual-gradient algorithms perform gradient descent on a single, unchanging error function: the mean squared Bellman residual error. Because of this, residual-

gradient algorithms are always guaranteed to converge in the same sense that pure backpropagation is guaranteed to converge. Epoch-wise training on a finite training set with a sufficiently small learning rate guarantees finding a solution that is arbitrarily close to a local minimum. Training on an infinite training set will converge to a local minimum if the training examples are drawn randomly from some distribution, weights change after each training example, and the learning rate decreases at an appropriate rate. A residual-gradient algorithm performs simple gradient descent. It can therefore be improved through the same methods that are used to improve backpropagation, such as adding momentum, or using pseudo-Newton, conjugate gradient, or heuristic second-order methods. Stochastic algorithms for supervised learning have been proposed that approximate the gradient during supervised learning rather than calculating it exactly (Barto, 1986) (Williams, 1992). This approach appears useful for hardware implementation, and residual-gradient algorithms could be implemented similarly.

Some residual-gradient algorithms have an additional property. Given any initial functions with nonzero Bellman residual, the residual can always be reduced slightly by infinitesimal changes to the functions. There are therefore no nonoptimal, local minima of the Bellman residual. This property of the Bellman equation ensures that if the function-approximation system is general enough to represent all possible functions, and if small changes in the functions correspond to small changes in the weights, then there are no nonoptimal, local minima for the weights. In this case, not only is the residual-gradient algorithm guaranteed to converge, it is guaranteed to converge to the optimal solution. This suggests that in general, it may be possible to ensure that a network will converge to arbitrarily small expected squared residuals if the network is a single-hidden-layer, sigmoidal perceptron with sufficiently many hidden units.

If a non-deterministic system is being controlled, it becomes necessary to generate two independent successors for a given state-action pair. When a model of the world is available, two successor states can simply be generated. When a model is not available, successor states can be generated in one of two ways. Experience can be cached in some manner and then samples can be chosen randomly. However, the algorithms presented here are being used with continuous-time problems and, thus, the number of states is potentially infinite. For this reason, storing all experience is not practical. An alternative is to learn the plant model on-line (Baird, 1995b).

3 THE *Q*-LEARNING ALGORITHM

3.1 *Q*-LEARNING (Non-residual-gradient form)

Q-learning is a reinforcement learning algorithm that is incremental (the weights are changed after each state transition), direct (a model of the MDP is not needed), and guaranteed to converge, at least for the discrete case with a finite number of states and actions. Furthermore, *Q*-learning can learn from any sequence of experiences in which every action is tried in every state infinitely often. Instead of storing values and policies, *Q*-learning stores *Q values*. For a given state x and action u, the optimal Q value, $Q^*(x,u)$, is the expected total discounted reinforcement that is received by starting in state x, performing action u on the first time step, then performing optimal actions thereafter. The maximum Q value in a state is the value of that state. The action associated with the maximum Q value in a state is the policy for that state. Initially, all Q values are set to arbitrary numbers. After an action u is performed in state x, the result is observed and the Q value is updated:

$$Q(x,u) \xleftarrow{\alpha} R_{\Delta t}(x,u) + \gamma^{\Delta t} \max_{u} Q(x',u) \tag{9}$$

The equivalent of the Bellman equation for *Q*-learning is:

$$Q(x,u) = R_{\Delta t}(x,u) + \gamma^{\Delta t} \sum_{x'} P(u,x,x') \max_{u'} Q(x',u') \tag{10}$$

The optimal Q function, $Q^*(x,u)$, is the unique solution to equation (10). The policies implied by Q^*, policies that always choose actions that maximize Q^*, are optimal policies.

Update (9) does not require a model of the MDP, nor does it contain any summations or integrals. The computational complexity of a single update is independent of the number of states. If the Q values are stored in a lookup table, then the complexity is linear in the number of actions, due to the time that it takes to find the maximum. However, the term being maximized is a stored function, not a calculated expression. This suggests that if Q is stored in an appropriate function approximation system, it might be possible to reduce even this part of the update to a constant-time algorithm. One algorithm that does this is described in Baird (1992). Another method, *wire fitting*, is described in Baird and Klopf (1993a). In both cases, the maximization of the function is performed incrementally during learning, rather than requiring an

exhaustive search for each update. Reinforcement learning systems based on discrete *Q*-learning are described in Baird and Klopf (1993b).

3.2 RESIDUAL-GRADIENT *Q*-LEARNING

The update rule for *Q*-learning, Equation (9), is guaranteed to converge to optimality when used with a lookup table, but when combined with a function-approximation system the *Q* function may not converge. The equivalent of the Bellman equation for *Q* learning is given in equation (10), where x' is the state reached by performing action u in state x. The Bellman residual for *Q*-learning is:

$$E(x,u) = \left(R(x,u) + \max_{u'} \gamma Q(x',u')\right) - Q(x,u) \tag{11}$$

The residual-gradient algorithm corresponding to *Q* -learning is:

$$\begin{aligned}
\Delta W &= -\frac{\alpha}{2}\frac{\partial E^2(x,u)}{\partial W} \\
&= -\alpha E(x,u)\frac{\partial E(x,u)}{\partial W} \\
&= -\alpha\left(\max_u\left(R(x,u) + \gamma Q(x',u)\right) - Q(x,u)\right) \\
&\qquad \left(\frac{\partial \max_u\left(\gamma Q(x',u)\right)}{\partial W} - \frac{\partial Q(x,u)}{\partial W}\right)
\end{aligned} \tag{12}$$

As with *Q*-learning, the corresponding residual-gradient algorithm does not contain a summation or integral, and so is not computationally intensive. Neither does it require a model be learned. *Q*-learning is guaranteed to converge to optimality with a lookup table, but not with an arbitrary, differentiable, function-approximation system. The residual-gradient algorithm is guaranteed to converge in both cases.

3.3 *Q*-LEARNING IN CONTINUOUS TIME

Q-learning requires relatively little computation per update, but it is useful to consider how the number of updates required scales with noise or with the duration of a time step, Δt. An important consideration is the relationship between *Q* values for the same state, and between *Q* values for the same action. The *Q* values $Q(x,u_1)$ and $Q(x,u_2)$ represent the long-term reinforcement received when starting in state x and performing

action u_1 or u_2 respectively, followed by optimal actions thereafter. In a typical reinforcement learning problem with continuous states and actions, it is frequently the case that performing one wrong action in a long sequence of optimal actions will have little effect on the total reinforcement. In such a case, $Q(x,u_1)$ and $Q(x,u_2)$ will have relatively close values. On the other hand, the values of widely separated states will typically not be close to each other. Therefore $Q(x_1,u)$ and $Q(x_2,u)$ may differ greatly for some choices of x_1 and x_2. The policy implied by a Q function is determined by the relative Q values in a single state. If the Q function is stored in a function approximation system with some error, the implied policy will tend to be sensitive to that error. As the time step duration Δt approaches zero, the penalty for one wrong action in a sequence decreases, the Q values for different actions in a given state become closer, and the implied policy becomes even more sensitive to noise or function approximation error. In the limit, for continuous time, the Q function contains no information about the policy. Therefore, Q-learning would be expected to learn slowly when the time steps are of short duration, due to the sensitivity to errors, and Q-learning is incapable of learning in continuous time. This problem is not a property of any particular function approximation system; rather, it is inherent in the definition of Q values.

4 THE ADVANTAGE UPDATING ALGORITHM

4.1 ADVANTAGE UPDATING (Non-residual-gradient form)

Reinforcement learning in continuous time is possible through the use of *advantage updating*. The advantage updating algorithm (Baird, 1993) is a reinforcement learning algorithm in which two types of information are stored. For each state x, the value, $V(x)$, is stored, representing an estimate of the total discounted return expected when starting in state x and performing optimal actions. For each state x and action u, the *advantage*, $A(x,u)$, is stored, representing an estimate of the degree to which the expected total discounted reinforcement is increased by performing action u rather than the action currently considered best. The optimal value function, $V^*(x)$, represents the true value of each state. The optimal advantage function, $A^*(x,u)$, will be zero if u is the optimal action (because u confers no advantage relative to itself) and $A^*(x,u)$ will be negative for any suboptimal u (because a suboptimal action has a negative advantage relative to the best action). The optimal advantage function, A^*, can be defined in terms of the optimal value function, V^*:

$$A^*(x,u) = \frac{1}{\Delta t}\left[R_{\Delta t}(x,u) - V^*(x) + \gamma^{\Delta t} V^*(x')\right] \tag{13}$$

The definition of an advantage includes a $1/\Delta t$ term to ensure that, for a small time step duration Δt, the advantages will not all go to zero. Advantages are related to Q values by:

$$A^*(x,u) = \frac{1}{\Delta t}\left[Q^*(x,u) - \max_{u'} Q^*(x,u')\right] \tag{14}$$

Both the value function and the advantage function are needed during learning, but after convergence to optimality, the policy can be extracted from the advantage function alone. The optimal policy for state x is any u that maximizes $A^*(x,u)$. The notation

$$A_{\max}(x) = \max_u A(x,u) \tag{15}$$

defines $A_{\max}(x)$. If $A_{\max}$ is zero in every state, then the advantage function is said to be *normalized*. $A_{\max}$ should eventually converge to zero in every state. The update rules for advantage updating in discrete time are as follows:

LEARN: perform action u_t in state x_t

$$A(x_t,u_t) \xleftarrow{\alpha} A_{\max}(x_t) + \frac{R_{\Delta t}(x_t,u_t) + \gamma^{\Delta t} V(x_{t+\Delta t}) - V(x_t)}{\Delta t} \tag{16}$$

$$V(x_t) \xleftarrow{\beta} V(x_t) + \left[A_{\max_{\text{new}}}(x_t) - A_{\max_{\text{old}}}(x_t)\right] / \alpha \tag{17}$$

NORMALIZE: pick an arbitrary state x and pick an action u randomly with uniform probability

$$A(x,u) \xleftarrow{\omega} A(x,u) - A_{\max}(x) \tag{18}$$

For the learning updates, the system performs action u_t in state x_t and observes the reinforcement received, $R_{\Delta t}(x_t,u_t)$, and the next state, $x_{t+\Delta t}$. The advantage and value functions are then updated according to updates (16) and (17). Update (16) modifies the advantage function $A(x,u)$. The maximum advantage in state x prior to applying update (16) is $A_{\max_{old}}(x)$. After applying update (16) the maximum is $A_{\max_{new}}(x)$. If these are different, then update (17) changes the value $V(x)$ by a proportional amount. As α goes to zero, the change in $A_{\max}$ goes to zero, but the change in $A_{\max}$ in update (17) is divided by α, so the value function will continue to learn at a reasonable rate as α decreases. Advantage updating can be applied to continuous-time systems by taking the

limit as Δt goes to zero. Substituting equation (5) into update (16) and taking the limit as Δt goes to zero yields:

$$A(x_t,u_t) \xleftarrow{\alpha} A_{\max}(x_t) + V(x_t)\ln\gamma + \cancel{V}(x_t) + r(x_t,u_t) \tag{19}$$

The learning updates, (16), (17), and (19), require interaction with the MDP or a model of the MDP, but the normalizing update, (18), does not. Normalizing updates can always be performed by evaluating and changing the stored functions independent of the MDP. Normalization is done to ensure that after convergence $A_{\max}(x)=0$ in every state. This avoids the representation problem noted above for Q-learning, where the Q function differs greatly between states but differs little between actions in the same state. Learning and normalizing can be performed asynchronously. For example, a system might perform a learning update once per time step, in states along a particular trajectory through state space, and perform a normalizing update multiple times per time step in states scattered randomly throughout the state space. The advantage updating algorithm is referred to as "advantage updating" rather than "advantage learning" because it includes both learning and normalizing updates.

The equivalent of the Bellman equation for advantage updating is a pair of simultaneous equations:

$$V(x) + A(x,u)\Delta t = R_{\Delta t}(x,u) + \gamma^{\Delta t}\sum_{x'} P(u,x,x')V(x') \tag{20}$$

$$\max_u A(x,u) = 0 \tag{21}$$

The unique solution to this set of equations is the optimal value and advantage functions $V^*(x)$ and $A^*(x,u)$. This can be seen by considering an arbitrary state x and the action $u_{\max}$ that maximizes the advantage in that state. For a given state, if (20) is satisfied, then the action that maximizes A will also maximize the right side of (20). If the advantage function satisfies (21), then $A(x,u_{\max})=0$. Equation (20) then reduces to equation (14), which is the Bellman equation. The only solution to this equation is $V=V^*$, so V^* is the unique solution to equations (20) and (21). Given that $V=V^*$, equation (20) can be solved for A , yielding equation (13), so the unique solution to the set of equations (20) and (21) is the pair of functions A^* and V^*.

The pair of equations (20) and (21) has the same unique solution as the pair (22) and (23), because equation (23) ensures that $A_{\max}(x)$ is zero in every state.

$$V(x) + \big(A(x,u) - A_{\max}(x)\big)\Delta t = R_{\Delta t}(x,u) + \gamma^{\Delta t} \sum_{x'} P(u,x,x')V(x') \quad (22)$$

$$\max_u A(x,u) = 0 \quad (23)$$

If, in state x, a large constant were added to each advantage $A^*(x,u)$ and to the value $V^*(x)$, then the resulting advantage and value functions would still satisfy equation (22). However, the advantage function would not satisfy equation (23), and so would be referred to as an *unnormalized* advantage function. Such a function would still be useful, because the optimal policy can be calculated from it, but it could be difficult to represent in a function approximation system. The learning updates (16) and (17) find value and advantage functions that satisfy (22). The normalizing update (18) ensures that the advantage function will be normalized, and so will satisfy (23) as well.

The update rules for advantage updating have a significant property: there are time derivatives in the update rules, but no gradients or partial derivatives. At time t, it is necessary to know $A_{\max}(x_t)$ and the value and rate of change of $V(x_t)$ while performing the current action. It is not necessary to know the partial derivative of V or A with respect to state or action. Nor is it necessary to know the partial derivative of next state with respect to current state or action. Only a few of the recent values of V need to be known in order to calculate the time derivative; there is no need for models of the system being controlled. Existing methods for solving continuous-time optimization problems, such as value iteration or differential dynamic programming, require that models be known or learned, and that partial derivatives of models be calculated. For a stochastic system controlled by a continuum of actions, previous methods also require maximizing over a set of one integral for each action. Q-learning improved on value iteration by not requiring the calculation of an integral during each update operation. Rather, Q-learning is incrementally performing a large number of integrals by a single stochastic sample on each time step. Advantage updating retains this desirable property. Advantage updating does not require the calculation of an integral during each update operation, and maximization is only done over stored values. For this reason, advantage updating appears useful for controlling stochastic systems.

4.2 RESIDUAL-GRADIENT ADVANTAGE UPDATING

The Bellman residuals for equations (20) and (21) are:

$$E_1(x_t,u) = \left(R(x_t,u) + \gamma^{\Delta t} V(x_{t+\Delta t}) - V(x_t)\right)\frac{1}{\Delta t} - A(x_t,u) + \max_{u'} A(x_t,u') \quad (24)$$

$$E_2(x,u) = -\max_u A(x,u) \quad (25)$$

Two equations exist instead of one, so the residual-gradient algorithm must perform gradient descent on the sum of the two squared Bellman residuals. If a function approximation system is used for the advantage and value functions, and if the system being controlled is deterministic, then, for incremental learning, a given weight W in the function-approximation system could be changed according to equation (26) on each time step:

$$\begin{aligned}
\Delta W &= -\frac{\alpha}{2}\frac{\partial\left[E_1^2(x_t,u_t) + E_2^2(x_t,u_t)\right]}{\partial W} \\
&= -\alpha E_1(x_t,u_t)\frac{\partial E_1(x_t,u_t)}{\partial W} - \alpha E_2(x_t,u_t)\frac{\partial E_2(x_t,u_t)}{\partial W} \\
&= -\alpha\left(\frac{1}{\Delta t}\left(R + \gamma^{\Delta t} V(x_{t+\Delta t}) - V(x_t)\right) - A(x_t,u_t) + \max_u A(x_t,u)\right) \\
&\quad \bullet\left(\frac{1}{\Delta t}\left(\gamma^{\Delta t}\frac{\partial V(x_{t+\Delta t})}{\partial W} - \frac{\partial V(x_t)}{\partial W}\right) - \frac{\partial A(x_t,u_t)}{\partial W} + \frac{\partial \max_u A(x_t,u)}{\partial W}\right) \\
&\quad -\alpha \max_u A(x_t,u)\frac{\partial \max_u A(x_t,u)}{\partial W}
\end{aligned} \quad (26)$$

As a simple gradient-descent algorithm, equation (26) is guaranteed to converge to the correct answer for a deterministic MDP, in the same sense that backpropagation is guaranteed to converge. However, if the MDP is nondeterministic, then it is necessary to generate two different possible "next states", $x_{t+\Delta t}$, for a given action, u_t, performed in a given state, x_t. One $x_{t+\Delta t}$ must be used to evaluate $V(x_{t+\Delta t})$, and the other must be used to evaluate $\frac{\partial}{\partial W} V(x_{t+\Delta t})$. This ensures that the weight change is an unbiased estimator of the true Bellman-residual gradient, but requires the system to generate a second $x_{t+\Delta t}$.

5 DIFFERENTIAL GAMES

Differential games (Isaacs, 1965) are played in continuous time, or use sufficiently small time steps to approximate continuous time. Both players evaluate the

given state and simultaneously execute an action, with no knowledge of the other player's selected action.

The *value* of a game is the long-term, discounted reinforcement if both opponents play the game optimally in every state. Consider a game in which player A tries to minimize the total discounted reinforcement, while the opponent, player B, tries to maximize the total discounted reinforcement. Given the advantage $A(x,u_A,u_B)$ for each possible action in state x, it is useful to define the minimax and maximin values for state x as:

$$\text{minimax}(x) = \min_{u_A} \max_{u_B} A(x, u_A, u_B) \tag{27}$$

$$\text{maximin}(x) = \max_{u_B} \min_{u_A} A(x, u_A, u_B) \tag{28}$$

If the minimax equals the maximin, then the minimax is called a *saddlepoint* and the optimal policy for both players is to perform the actions associated with the saddlepoint. If a saddlepoint does not exist, then the optimal policy is stochastic if an optimal policy exists at all. If a saddlepoint does not exist, and a learning system treats the minimax as if it were a saddlepoint, then the system will behave as if player A must choose an action on each time step, and then player B chooses an action based upon the action chosen by A. For the algorithms described below, a saddlepoint is assumed to exist. If a saddlepoint does not exist, this assumption confers a slight advantage to player B. Our system is a differential game with a missile pursuing a plane, as in Rajan, Prasad, and Rao (1980) and Millington (1991). The action performed by the missile is a function of the state. The action performed by the plane is a function of the state and the action of the missile. Because we assume a saddle point exists, the knowledge the plane has of the missile's action offers no benefit to the plane. Furthermore, the use of the minimax for determination of policy guarantees a solution to the game by ensuring a deterministic system.

6 SIMULATED MISSILE-AIRCRAFT DIFFERENTIAL GAME

6.1 GAME DEFINITION

A linear-quadratic, differential game was employed for comparing Q-learning to advantage updating in both the residual and non-residual-gradient forms. The game has two players, a missile and a plane. The state $\mathbf{x}$ is a vector $(\mathbf{x}_m,\mathbf{x}_p)$ composed of the state

of the missile and the state of the plane, each of which are composed of the position and velocity of the player in two-dimensional space. The action **u** is a vector ($\mathbf{u}_m$,$\mathbf{u}_p$) composed of the action performed by the missile and the action performed by the plane, each of which are the acceleration of the player in two-dimensional space. The dynamics of the system are linear; the next state, $\mathbf{x}_{t+1}$, is a linear function of the current state, $\mathbf{x}_t$, and action, $\mathbf{u}_t$. The dynamics are defined explicitly by the following:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_p \\ \mathbf{x}_m \\ \dot{\mathbf{x}}_p \\ \dot{\mathbf{x}}_m \end{bmatrix} \qquad \mathbf{U} = \begin{bmatrix} \mathbf{u}_p \\ \mathbf{u}_m \end{bmatrix} \tag{29}$$

$$\mathbf{X}_{t+1} = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \bullet \mathbf{X}_t + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ \Delta t & 0 \\ 0 & \Delta t \end{bmatrix} \bullet \mathbf{U}_t \tag{30}$$

The reinforcement function, *R*, is a quadratic function of the accelerations and the distance between the players.

$$R(x,u) = [\text{distance}^2 + (\text{missile acceleration})^2 - 2(\text{plane acceleration})^2]\Delta t \tag{31}$$

$$R(\mathbf{x},\mathbf{u}) = \left[\left(\mathbf{x}_m - \mathbf{x}_p\right)^2 + \mathbf{u}_m^2 - 2\mathbf{u}_p^2\right]\Delta t \tag{32}$$

In equation (32), squaring a vector is equivalent to taking the dot product of the vector with itself. The missile seeks to minimize the reinforcement, and the plane seeks to maximize reinforcement. The plane receives twice as much punishment for acceleration as does the missile, thus allowing the missile to accelerate twice as easily as the plane.

The value function *V* is a quadratic function of the state. In equation (33), $\mathbf{D_m}$ and $\mathbf{D_p}$ are weight matrices that change during learning.

$$V(\mathbf{x}) = \mathbf{x}_m^T D_m \mathbf{x}_m + \mathbf{x}_p^T D_p \mathbf{x}_p \tag{33}$$

The advantage function *A* is a quadratic function of the state **x** and action **u**. The actions are accelerations of the missile and plane in two dimensions.

$$A(\mathbf{x},\mathbf{u}) = \mathbf{x}_m^T\mathbf{A}_m\mathbf{x}_m + \mathbf{x}_m^T\mathbf{B}_m\mathbf{C}_m\mathbf{u}_m + \mathbf{u}_m^T\mathbf{C}_m\mathbf{u}_m + \mathbf{x}_p^T\mathbf{A}_p\mathbf{x}_p + \mathbf{x}_p^T\mathbf{B}_p\mathbf{C}_p\mathbf{u}_p + \mathbf{u}_p^T\mathbf{C}_p\mathbf{u}_p \tag{34}$$

The matrices **A**, **B**, and **C** are the adjustable weights that change during learning. Equation (34) is the sum of two general quadratic functions. This would still be true if the second and fifth terms were **xBu** instead of **xBCu**. The latter form was used to simplify the calculation of the policy. Using the **xBu** form, the gradient is zero when $\mathbf{u}=-\mathbf{C}^{-1}\mathbf{B}\mathbf{x}/2$. Using the **xBCu** form, the gradient of $A(\mathbf{x},\mathbf{u})$ with respect to **u** is zero when $\mathbf{u}=-\mathbf{B}\mathbf{x}/2$, which avoids the need to invert a matrix while calculating the policy.

6.2 THE BELLMAN RESIDUAL AND UPDATE EQUATIONS

Equations (24) and (25) define the Bellman residuals when maximizing the total discounted reinforcement for an optimal control problem. Equation (35) is the equivalent of advantage updating for games, so the $\max_u$ term is replaced by minimax, as defined in equation (27). This modification allows the use of the equations for differential game control, rather than optimal control. The same substitutions were made in the *Q*-learning algorithms as well.

$$E_1(x_t,u_t) = \left(R(x_t,u_t) + \gamma^{\Delta t}V(x_{t+\Delta t}) - V(x_t)\right)\frac{1}{\Delta t} - A(x_t,u_t) + minimax\ A(x_t) \tag{35}$$

$$E_2(x_t,u_t) = -minimax\ A(x_t) \tag{36}$$

The resulting weight update equation is:

$$\Delta W = -\alpha\left(\left(R + \gamma^{\Delta t}V(x_{t+\Delta t}) - V(x_t)\right)\frac{1}{\Delta t} - A(x_t,u_t) + minimax\ A(x_t)\right) \bullet\left(\left(\gamma^{\Delta t}\frac{\partial V(x_{t+\Delta t})}{\partial W} - \frac{\partial V(x_t)}{\partial W}\right)\frac{1}{\Delta t} - \frac{\partial A(x_t,u_t)}{\partial W} + \frac{\partial minimax\ A(x_t)}{\partial W}\right) - \alpha minimax\ A(x_t)\frac{\partial minimax\ A(x_t)}{\partial W} \tag{37}$$

The training procedure for the network is as follows:

(1) Initialize the state vector, $\mathbf{x}_t$, and action vector, $\mathbf{u}_t$, to random values in the range [0,1];

(2) Calculate the *minimax* for the given state;

(3) Calculate $V(\mathbf{x}_t)$;

(4) Calculate $A(\mathbf{x}_t,\mathbf{u}_t)$;

(5) Calculate $R(\mathbf{x}_t,\mathbf{u}_t)$;

(6) Transition from state $\mathbf{x}_t$ to $\mathbf{x}_{t+\Delta t}$;

(7) Calculate $V(\mathbf{x}_{t+\Delta t})$;

(8) Update the weights according to equation (37).

For the simulations described here, a new state is chosen in each time step during training . When controlling a physical system, one could train on the states in the order they are visited, or a set of state transitions could be stored in a buffer and used for training in a random order.

7 RESULTS

Experiments were formulated to accomplish two objectives. The first objective was to determine to what degree advantage updating could learn the optimal policy for the missile/aircraft system. The second objective was to assess the utility of advantage updating relative to *Q*-learning in approximated continuous-time systems.

7.1 INITIAL RESULTS FOR ADVANTAGE UPDATING

Experiment 1, described below, provides proof-of-concept for the residual-gradient form of advantage updating and accomplishes part of the first objective. Experiment 1 demonstrates that advantage updating is able to learn the optimal policy for the given system.

7.1.1 Experiment 1

The optimal weight matrices $\mathbf{A}^*$, $\mathbf{B}^*$, $\mathbf{C}^*$, and $\mathbf{D}^*$ were calculated numerically using *Mathematica*. This was possible because of the linear-quadratic nature of the problem. These matrices were then used in determining the ability of the algorithm to converge to the optimal answer.

The residual-gradient form of advantage updating learned the correct policy weights to three significant figures after approximately three days of training on a Macintosh Quadra 950. The missile learned to pursue the plane, and the plane learned to evade the missile (Figure 1). Interesting behavior was exhibited by the plane under

certain initial conditions. First, the plane learned that in some cases it is better to turn toward the missile in the short term to increase the distance between the two in the long term (Figure 2), a tactic sometimes used by pilots. Second, the missile learned to lead the plane rather than simply homing in on a heat source or radar signature (Figure 3).

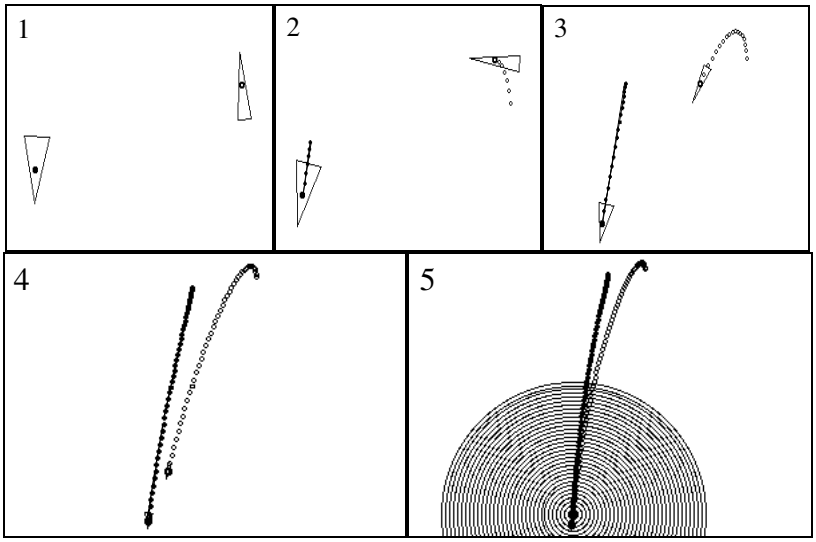

Figure 1: Progression from initial state to missile impact. The plane is represented by the thick triangle and leaves a connected path. The missile is represented by the thin triangle and leaves a dotted path. Each successive diagram displays a larger geographical area, explaining why aircraft and missile size varies.

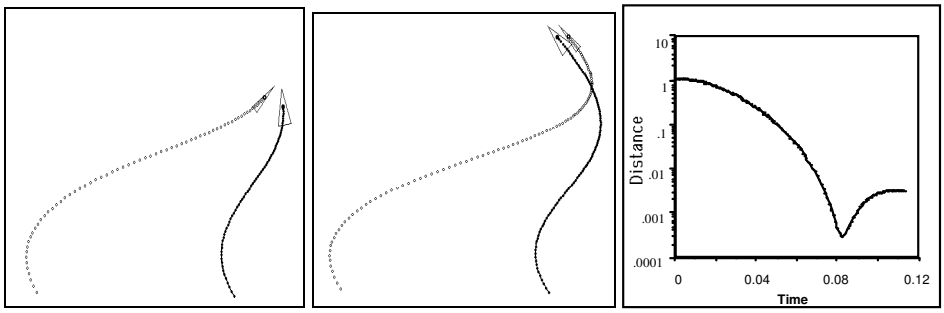

Figure 2: An example of the plane turning toward the missile in the short term in order to increase the distance between the plane and missile in the long term.

Included is a graph of distance versus time showing the effects of the plane's decision.

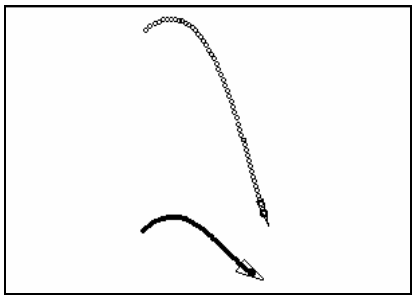

Figure 3: After learning, the missile leads the plane, rather then simply homing in on a heat source or radar signature.

7.2 COMPARATIVE ASSESSMENT

The purpose of Experiment 2, described below, is to assess the utility of advantage updating relative to *Q*-learning. Experiment 3 extends the results of Experiment 2 to provide a more complete comparison of the four tested algorithms.

7.2.1 Experiment 2

The error in the policy of a learning system is defined to be the sum of the squared errors in the **B** matrix weights. The optimal policy weights in this problem are the same for both advantage updating and *Q*-learning, so this metric can be used to compare results for both algorithms. Four different learning algorithms were compared: advantage updating, *Q*-learning, residual-gradient advantage updating, and residual-gradient *Q*-learning. No results are shown for advantage updating in the non-residual-gradient form because this algorithm was unstable to the point that no meaningful results could be obtained; the weights always grew to infinity.

Experiment 2 was performed for four different time step durations: 0.05, 0.005, 0.0005, and 0.00005. The learning rates for both forms of *Q*-learning were optimized to one significant figure for each simulation. A single learning rate was used for residual-gradient advantage updating in all four simulations. It is possible that advantage updating would have performed better with different learning rates. For each algorithm, the error was calculated after learning for 40,000 iterations. The process was repeated 10 times using different random number seeds and the results were averaged.

The non-residual-gradient form of *Q*-learning appeared to work better when the weights were initialized to small numbers. Therefore, the initial weights were chosen randomly between 0 and 1 for the residual-gradient forms of the algorithms, and between 0 and 10^{-8} for the non-residual-gradient form of *Q*-learning. For small time steps, non-residual-gradient *Q*-learning performed so poorly that the error was lower for a learning

rate of zero (no learning) than it was for a learning rate of 10^{-8}. Table 1 gives the learning rates used for each simulation, and Figure 4 shows the resulting error after learning.

	Time step duration, Δt			
	$5 \cdot 10^{-2}$	$5 \cdot 10^{-3}$	$5 \cdot 10^{-4}$	$5 \cdot 10^{-5}$
Q	0.02	0.06	0.2	0.4
RQ	0.08	0.09	0	0
RAU	0.005	0.005	0.005	0.005

Table 1: Learning rates used for each simulation. Learning rates are optimal to one significant figure for both forms of *Q*-learning, but are not necessarily optimal for advantage updating. (Q:*Q*-learning, RQ: residual-gradient *Q*-learning, RAU: residual-gradient advantage updating)

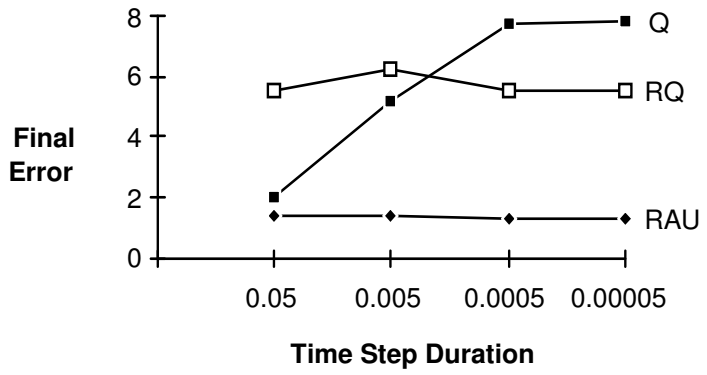

Figure 4: Comparison of error versus time step duration for *Q*-Learning (Q), residual-gradient *Q*-Learning (RQ), and residual-gradient advantage updating (RAU) using rates optimal to one significant figure for both forms of *Q*-learning and not optimized for advantage updating. The final error is the sum of squared errors in the **B** matrix weights after 40,000 time steps of learning. The residual-gradient form of advantage updating yielded a lower error than either form of *Q*-learning in every case, with the relative improvement of performance increasing for smaller time step durations.

7.2.2 Experiment 3

The objective in this experiment was to train each learning system until convergence. These systemns learned for many more iterations than the systems in Experiment 2. The learning rates used for this experiment were identical to the rates that were found to be optimal for the shorter runs (see Table 1).

The weights for the non-residual-gradient algorithms grew without bound in all of these extended experiments, regardless of the learning rates. Figure 5 compares the algorithms' abilities to converge to the correct policy. The total squared error in each algorithm's policy weights is plotted as a function of learning time. Residual-gradient advantage updating was able to learn the optimal policy while *Q*-learning was unable to learn a policy that was better than that of the initial, randomly chosen weights.

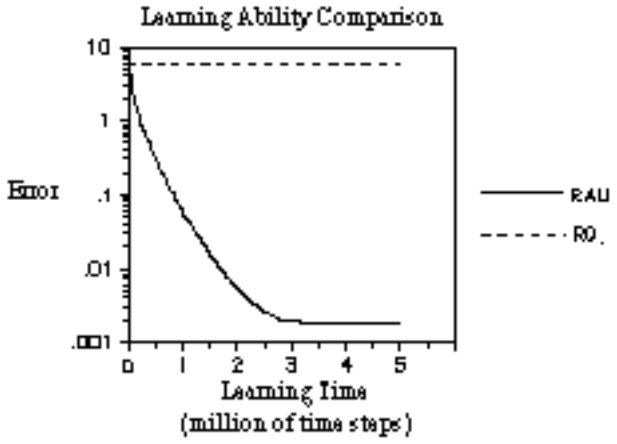

Figure 5

8 CONCLUSIONS

Residual-gradient advantage updating learned the optimal policy for both the plane and the missile. The system demonstrated interesting behavior in that the plane learned to maximize the distance between itself and the missile in the long term by causing the distance to decrease in the short term.

The experimental data shows that residual-gradient advantage updating yields better performance in all cases than the other three algorithms tested. As the time steps grow shorter, *Q*-learning becomes slower in learning the optimal policy, while advantage

updating avoids this difficulty, reducing the error at a constant rate regardless of the time step duration. In many cases, the non-residual-gradient forms of Q-learning and advantage updating were unstable to the point that no meaningful results could be obtained; the weights grew without bound. When training the system to convergence, only the weights of the residual-gradient forms of the algorithms did not grow to infinity. The residual-gradient form of Q-learning was unable to learn a policy that was any better than that of the initial, randomly chosen weights, while residual-gradient advantage updating learned the optimal policy weights to two significant figures.

The residual-gradient form of advantage updating appears to combine the desirable features of Q-learning and residual-gradient algorithms in general. Residual-gradient advantage updating is guaranteed to converge with general function-approximation systems. It does not require a model to be learned and can be implemented with relatively little computation per update. It learns functions that are more robust to noise and small time steps than are Q functions and is simpler than non-residual-gradient advantage updating.

8.1 FUTURE RESEARCH

The implementations of reinforcement learning systems recounted here employed quadratic function approximators to store the value, advantage, and Q functions for the advantage updating and Q-learning algorithms. In doing so, it became easy to determine the action associated with the policy of the system at any time t. Future research will include the use of more general function approximation systems (e.g., radial-basis functions, sigmoidal networks, memory-based interpolation). This leads to an open question in reinforcement learning: how can the action that is associated with the maximum Q value or advantage be chosen from a continuous range of actions? An algorithm called *wire-fitting*, developed by Baird and Klopf (1993a), offers one possible approach and is planned for future testing.

Acknowledgment

This research was supported under Task 2312R1 by the United States Air Force Office of Scientific Research.

References

Baird, L. C. (1992). Function minimization for dynamic programming using connectionist networks. *Proceedings of the IEEE Conference on Systems, Man, and Cybernetics* (pp. 19-24). Chicago, IL.

Baird, L. C. (1993). *Advantage updating* (DTIC Report AD WL-TR-93-1146, available from the Defense Technical Information Center, Cameron Station, Alexandria, VA 22304-6145). Wright-Patterson Air Force Base, OH.

Baird, L. C. (1995a). Residual algorithms: reinforcement learning with function approximation. In Armand Prieditis & Stuart Russell (Eds.), *Machine Learning: Proceedings of the Twelfth International Conference*. San Francisco, CA: Morgan Kaufman Publishers.

Baird, L. C. (1995b). *ANN emulations of arbitrary PDFs* (Department of Computer Science technical report to be published). Colorado Springs, CO: United States Air Force Academy.

Baird, L. C., & Klopf, A. H. (1993a). *Reinforcement learning with high-dimensional, continuous actions.* (DTIC Report AD WL-TR-93-1147, available from the Defense Technical Information Center, Cameron Station, Alexandria, VA 22304-6145). Wright-Patterson Air Force Base, OH.

Baird, L. C., & Klopf, A. H. (1993b). A hierarchical network of provably optimal learning control systems: Extensions of the associative control process (ACP) network. *Adaptive Behavior*, **1**(3), 321-352.

Bertsekas, D. P. (1987). *Dynamic programming: Deterministic and stochastic models.* Englewood Cliffs, NJ: Prentice-Hall.

Barto, A. G. (1986). Game-theoretic cooperativity in networks of self-interested units. In J.S. Denker (ed.), *Neural Networks For Computing*. New York: American Institute of Physics.

Bradtke, S. J. (1993). Reinforcement learning applied to linear quadratic regulation. *Proceedings of the Fifth Conference on Neural Information Processing Systems* (pp. 295-302). Morgan Kaufmann.

Isaacs, R. (1965). *Differential games*. New York: Wiley, Inc.

Millington, P. J. (1991). *Associative reinforcement learning for optimal control*. Unpublished master's thesis, Massachusetts Institute of Technology, Cambridge, MA.

Rajan, N., Prasad, U. R., and Rao, N. J. (1980). Pursuit-evasion of two aircraft in a horizontal plane. *Journal of Guidance and Control*, **3**(3), 261-267.

Rumelhart, D., Hinton, G., & Williams, R. J. (1986). Learning representations by backpropagating errors. *Nature*. **323**, 533-536.

Tesauro, G. (1990). Neurogammon: A neural-network backgammon program. *Proceedings of the International Joint Conference on Neural Networks* (pp. 33-40). San Diego, CA.

Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning*, **8(3/4)**, 279-292.

Watkins, C. J. C. H. (1989). *Learning from delayed rewards*. Doctoral thesis, Cambridge University, Cambridge, England.

Williams, R. J. (1992). Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning*, **8**, 229-256.

Williams, R. J., & Baird, L. C. (1993). *Analysis of some incremental variants of policy iteration: First steps toward understanding actor-critic learning systems.* (Tech. Rep. NU-CCS-93-11). Boston, MA: Northeastern University, College of Computer Science.

Appendix A: Notation

x_t	State at time t
u_t	Control action at time t. In discrete-time control, action is constant throughout a time step.
$r_{\Delta t}(x_t,u_t)$	Rate of reinforcement at time t while performing action u_t in state x_t.
$R_{\Delta t}(x,u)$	Total discounted reinforcement during a single time step starting in state x with constant action u.. R is the integral of r as time varies over a single time step.
$\pi^*(x)$	Optimal control action to perform in state x.
$V^*(x)$	Total discounted reinforcement over all time if starting in state x then acting optimally.
$Q^*(x,u)$	Total discounted reinforcement over all time if starting in state x, doing u, then acting optimally.
$A^*(x,u)$	Amount by which action u is better than the optimal action in maximizing total discounted reinforcement over all time. A^* is zero for optimal actions, negative for all other actions.
π,V,Q,A	Learning system's estimates of π^*, V^*, Q^*, and A^*.

All parameter updates are represented by arrows. When a parameter is updated during learning, the notation:

$$W \longleftarrow K \tag{A1}$$

represents the operation of instantaneously changing the parameter W so that its new value is K, whereas:

$$W \overset{\alpha}{\longleftarrow} K \tag{A2}$$

represents a partial movement of the value of W toward K, which is equivalent to:

$$W_{new} \longleftarrow (1-\alpha)W_{old} + \alpha K \tag{A3}$$

where the learning rate α is a small positive number.

Appendix B: Difficulties in combining function-approximation systems with reinforcement learning algorithms.

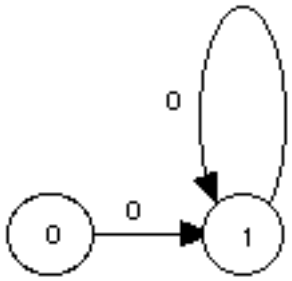

Figure B1

Figure 1 gives an example Markov chain with two states, named 0 and 1, and a deterministic transition from each state to state 1. Every transition results in a reinforcement of zero, so the optimal value function is $V^*(0)=V^*(1)=0$. The value function given in equation (B1), is a linear function with only one parameter to be learned, the weight W. Typical function-approximation systems use nonlinear functions with multiple weights, but even this linear function with a single weight is capable of exhibiting convergence difficulties,

$$V(x) = (1+4x)W \tag{B1}$$

It is possible for equation (B1) to represent the optimal function exactly; this occurs when $W=0$. Ideally, W would converge to zero during learning, but this is not always the case. If $\gamma=0.9$ and the weight change, ΔW, is calculated for each of the two possible transitions, and the sum of the two changes is used to modify the weight, then the total weight change is:

$$\begin{aligned}\Delta W_{total} &= \alpha\left[(0+\gamma V(1)-V(0))\frac{\partial V(0)}{\partial W}+(0+\gamma V(1)-V(1))\frac{\partial V(1)}{\partial W}\right] \\ &= \alpha\left[(0+0.9(1+4)W-W)1+(0+0.9(1+4)W-(1+4)W)(1+4)\right] \\ &= \alpha W \end{aligned} \tag{B2}$$

Note that if the weight is initialized to the correct answer, exactly zero, then the weight will remain zero. However, if the weight is initialized to any positive number, no matter how small, repeated applications of update (B2) will cause the weight to grow toward infinity, and if the weight is initially negative, it will grow toward negative infinity.