Reinforcement Learning: A Tutorial

Mance E. Harmon
WL/AACF
2241 Avionics Circle
Wright Laboratory
Wright-Patterson AFB, OH 45433
mharmon@acm.org

Stephanie S. Harmon
Wright State University
156-8 Mallard Glen Drive
Centerville, OH 45458

Scope of Tutorial

The purpose of this tutorial is to provide an introduction to reinforcement learning (RL) at a level easily understood by students and researchers in a wide range of disciplines. The intent is not to present a rigorous mathematical discussion that requires a great deal of effort on the part of the reader, but rather to present a conceptual framework that might serve as an introduction to a more rigorous study of RL. The fundamental principles and techniques used to solve RL problems are presented. The most popular RL algorithms are presented. Section 1 presents an overview of RL and provides a simple example to develop intuition of the underlying dynamic programming mechanism. In Section 2 the parts of a reinforcement learning problem are discussed. These include the environment, reinforcement function, and value function. Section 3 gives a description of the most widely used reinforcement learning algorithms. These include TD(λ) and both the residual and direct forms of value iteration, Q-learning, and advantage learning. In Section 4 some of the ancillary issues in RL are briefly discussed, such as choosing an exploration strategy and an appropriate discount factor. The conclusion is given in Section 5. Finally, Section 6 is a glossary of commonly used terms followed by references in Section 7 and a bibliography of RL applications in Section 8. The tutorial structure is such that each section builds on the information provided in previous sections. It is assumed that the reader has some knowledge of learning algorithms that rely on gradient descent (such as the backpropagation of errors algorithm).

1 Introduction

There are many unsolved problems that computers could solve if the appropriate software existed. Flight control systems for aircraft, automated manufacturing systems, and sophisticated avionics systems all present difficult, nonlinear control problems. Many of these problems are currently unsolvable, not because current computers are too slow or have too little memory, but simply because it is too difficult to determine what the program should do. If a computer could learn to solve the problems through trial and error, that would be of great practical value.

Reinforcement Learning is an approach to machine intelligence that combines two disciplines to successfully solve problems that neither discipline can address individually. *Dynamic Programming* is a field of mathematics that has traditionally been used to solve problems of optimization and control. However, traditional dynamic programming is limited in the size and complexity of the problems it can address.

Supervised learning is a general method for training a parameterized function approximator, such as a neural network, to represent functions. However, supervised learning requires sample input-output pairs from the function to be learned. In other words, supervised learning requires a set of questions with the right answers. For example, we might not know the best way to program a computer to recognize an infrared picture of a tank, but we do have a large collection of infrared pictures, and we do know whether each picture contains a tank or not. Supervised learning could look at all the examples with answers, and learn how to recognize tanks in general.

Unfortunately, there are many situations where we don't know the correct answers that supervised learning requires. For example, in a flight control system, the question would be the set of all sensor readings at a given time, and the answer would be how the flight control surfaces should move during the next millisecond. Simple neural networks can't learn to fly the plane unless there is a set of known answers, so if we don't know how to build a controller in the first place, simple supervised learning won't help.

For these reasons there has been much interest recently in a different approach known as *reinforcement learning* (RL). Reinforcement learning is not a type of neural network, nor is it an alternative to neural networks. Rather, it is an orthogonal approach that addresses a different, more difficult question. Reinforcement learning combines the fields of dynamic programming and supervised learning to yield powerful machine-learning systems. Reinforcement learning appeals to many researchers because of its generality. In RL, the computer is simply given a goal to achieve. The computer then learns how to achieve that goal by trial-and-error interactions with its environment. Thus, many researchers are pursuing this form of machine intelligence and are excited about the possibility of solving problems that have been previously unsolvable.

To provide the intuition behind reinforcement learning consider the problem of learning to ride a bicycle. The goal given to the RL system is simply to ride the bicycle without falling over. In the first trial, the RL system begins riding the bicycle and performs a series of actions that result in the bicycle being tilted 45 degrees to the right. At this point their are two actions possible: turn the handle bars left or turn them right. The RL system turns the handle bars to the left and immediately crashes to the ground, thus receiving a negative reinforcement. The RL system has just learned not to turn the handle bars left when tilted 45 degrees to the right. In the next trial the RL system performs a series of actions that again result in the bicycle being tilted 45 degrees to the right. The RL system knows not to turn the handle bars to the left, so it performs the only other possible action: turn right. It immediately crashes to the ground, again receiving a strong negative reinforcement. At this point the RL system has not only learned that turning the handle bars right or left when tilted 45 degrees to the right is bad, but that the "state" of being titled 45 degrees to the right is bad. Again, the RL system begins another trial and performs a series of actions that result in the bicycle being tilted 40 degrees to the right. Two actions are possible: turn right or turn left. The RL system turns the handle bars left which results in the bicycle being tilted 45 degrees to the right, and ultimately results in a strong negative reinforcement. The RL system has just learned not to turn the handle bars to the left when titled 40 degrees to the right. By performing enough of these trial-and-error interactions with the environment, the RL system will ultimately learn how to prevent the bicycle from ever falling over.

2 The Parts Of A Reinforcement Learning Problem

In the standard reinforcement learning model an agent interacts with its environment. This interaction takes the form of the agent sensing the environment, and based on this sensory input choosing an action to perform in the environment. The action changes the environment in some manner and this change is communicated to the agent through a scalar reinforcement signal. There are three fundamental parts of a reinforcement learning problem: the environment, the reinforcement function, and the value function.

The Environment

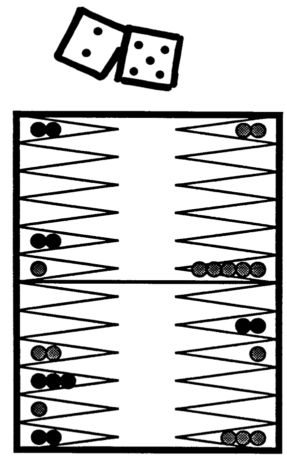
Every RL system learns a mapping from situations to actions by trial-and-error interactions with a dynamic environment. This environment must at least be partially observable by the reinforcement learning system, and the observations may come in the form of sensor readings, symbolic descriptions, or possibly "mental" situations (e.g., the situation of being lost). The actions may be low level (e.g., voltage to motors), high level (e.g., accept job offer), or even "mental" (e.g., shift in focus of attention). If the RL system can observe perfectly all the information in the environment that might influence the choice of action to perform, then the RL system chooses actions based on true "states" of the environment. This ideal case is the best possible basis for reinforcement learning and, in fact, is a necessary condition for much of the associated theory.

The Reinforcement Function

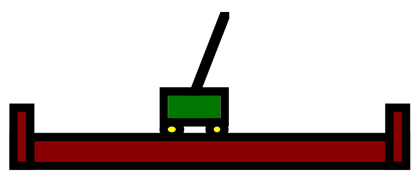
As stated previously, RL systems learn a mapping from situations to actions by trial-and-error interactions with a dynamic environment. The "goal" of the RL system is defined using the concept of a *reinforcement function*, which is the exact function of future reinforcements the agent seeks to maximize. In other words, there exists a mapping from state/action pairs to reinforcements; after performing an action in a given state the RL agent will receive some reinforcement (reward) in the form of a scalar value. The RL agent learns to perform actions that will maximize the sum of the reinforcements received when starting from some initial state and proceeding to a terminal state.

It is the job of the RL system designer to define a reinforcement function that properly defines the goals of the RL agent. Although complex reinforcement functions can be defined, there are at least three noteworthy classes often used to construct reinforcement functions that properly define the desired goals.

Pure Delayed Reward and Avoidance Problems

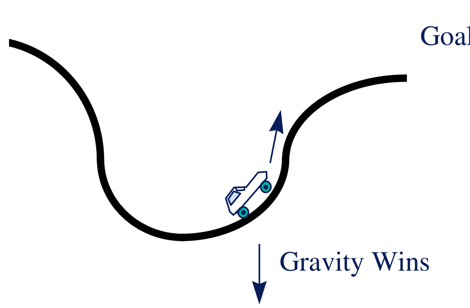


In the Pure Delayed Reward class of functions the reinforcements are all zero except at the terminal state. The sign of the scalar reinforcement at the terminal state indicates whether the terminal state is a goal state (a reward) or a state that should be avoided (a penalty). For example, if one wanted an RL agent to learn to play the game of backgammon, the system could be defined as follows. The situation (state) would be the configuration of the playing board (the location of each player's pieces). In this case there are approximately 10^{20} different possible states. The actions available to the agent are the set of legal moves. The reinforcement function is defined to be zero after every turn except when an action results in a win or a loss, in which case the agent receives a +1 reinforcement for a win, and a -1 reinforcement for a loss. Because the agent is trying to maximize the reinforcement, it will learn that the states corresponding to a win are goal states and states resulting in a loss are to be avoided.



Another example of a pure delayed reward reinforcement function can be found in the standard cart-pole or inverted pendulum problem. A cart supporting a hinged, inverted pendulum is placed on a finite track. The goal of the RL agent is to learn to balance the pendulum in an upright position without hitting the end of the track. The situation (state) is the dynamic state of the cart pole system. Two actions are available to the agent in each state: move the cart left, or move the cart right. The reinforcement function is zero everywhere except for the states in which the pole falls or the cart hits the end of the track, in which case the agent receives a -1 reinforcement. Again, because the agent is trying to maximize total reinforcement, the agent will learn the sequence of actions necessary to balance the pole and avoid the -1 reinforcement.

Minimum Time to Goal



Reinforcement functions in this class cause an agent to perform actions that generate the shortest path or trajectory to a goal state. An example is an experiment commonly known as the "Car on the hill" problem. The problem is defined as that of a stationary car being positioned between two steep inclines. The goal of the driver (RL agent) is to successfully drive up the incline on the right to reach a goal state at the top of the hill. The state of the environment is the car's position and velocity. Three actions are available to the agent in each state: forward thrust, backward thrust, or no thrust at all. The dynamics of the system are such that the car does not have enough thrust to simply drive up the hill. Rather, the driver must learn to use momentum to his advantage to gain enough velocity to successfully climb the hill. The reinforcement function is -1 for ALL state transitions except the transition to the goal state, in which case a zero reinforcement is returned. Because the agent wishes to maximize reinforcement, it learns to choose actions that minimize the time it takes to reach the goal state, and in so doing learns the optimal strategy for driving the car up the hill.

Games

Thus far it has been assumed that the learning agent always attempts to maximize the reinforcement function. This need not be the case. The learning agent could just as easily learn to minimize the reinforcement function. This might be the case when the reinforcement is a function of limited resources and the agent must learn to conserve these resources while achieving a goal (e.g., an airplane executing a maneuver while conserving as much fuel as possible).

An alternative reinforcement function would be used in the context of a game environment, when there are two or more players with opposing goals. In a game scenario, the RL system can learn to generate optimal behavior for the players involved by finding the maximin, minimax, or saddlepoint of the reinforcement function. For example, a missile might be given the goal of minimizing the distance to a given target (in this case an airplane). The airplane would be given the opposing goal of maximizing the distance to the missile. The agent would evaluate the state for each player and would choose an action independent of the other players action. These actions would then be executed in parallel.

Because the actions are chosen independently and executed simultaneously, the RL agent learns to choose actions for each player that would generate the best outcome for the given player in a "worst case" scenario. The agent will perform actions for the missile that will minimize the maximum distance to the airplane assuming the airplane will choose the action that maximizes the same distance. The agent will perform actions for the airplane that will maximize the minimum distance to the missile assuming the missile will perform the action that will minimize the same distance. A more detailed discussion of this alternative can be found in Harmon, Baird, and Klopf (1994), and Littman(1996).

The Value Function

In previous sections the environment and the reinforcement function are discussed. However, the issue of how the agent learns to choose "good" actions, or even how we might measure the utility of an action is not explained. First, two terms are defined. A *policy* determines which action should be performed in each state; a policy is a mapping from states to actions. The *value* of a state is defined as the sum of the reinforcements received when starting in that state and following some fixed policy to a terminal state. The optimal policy would therefore be the mapping from states to actions that maximizes the sum of the reinforcements when starting in an arbitrary state and performing actions until a terminal state is reached. Under this definition the value of a state is dependent upon the policy. The *value function* is a mapping from states to state values and can be approximated using any type of function approximator (e.g., multi-layered perceptron, memory based system, radial basis functions, look-up table, etc.).

An example of a value function can be seen using a simple Markov decision process with 16 states. The state space can be visualized using a 4x4 grid. Each square represents a state. The reinforcement function is -1 everywhere (i.e., the agent receives a reinforcement of -1 on each transition). There are 4 actions possible in each state: north, south, east, west. The goal states are the upper left corner and the lower right corner. The value function for the random policy is shown in Figure 1. For each state the random policy randomly chooses one of the four possible actions. The numbers in the states represent the expected values of the states. For example, when starting in the lower left corner and following a random policy, on average there will be 22 transitions to other states before the terminal state is reached.

0	-14	-20	-22
-14	-18	-22	-20
-20	-22	-18	-14
-22	-20	-14	0

Figure 1

The optimal value function is shown in Figure 2. Again, starting in the lower left corner, calculating the sum of the reinforcements when performing the optimal policy (the policy that will maximize the sum of the reinforcements), the value of that state is -3 because it takes only 3 transitions to reach a terminal state. If we are given the optimal value function, then it becomes a trivial task to extract the optimal policy. For example, one can start in any state in Figure 2 and simply choose the action that maximizes the immediate reinforcement received. In other words, one can perform a one level deep breadth-first search over actions to find the action that will maximize the immediate reward. The optimal policy for the value function shown in Figure 2 is given in Figure 3.

This leads us to the fundamental question of almost all of reinforcement learning research: How do we devise an algorithm that will efficiently find the optimal value function?

0	-1	-2	-3
-1	-2	-3	-2
-2	-3	-2	-1
-3	-2	-1	0

Figure 2

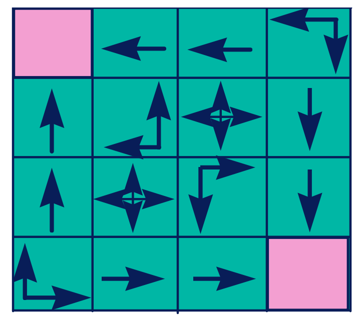


Figure 3

3 Approximating the Value Function

Reinforcement learning is a difficult problem because the learning system may perform an action and not be told whether that action was good or bad. For example, a learning auto-pilot program might be given control of a simulator and told not to crash. It will have to make many decisions each second and then, after acting on thousands of decisions, the aircraft might crash. What should the system learn from this experience? Which of its many actions were responsible for the crash? Assigning blame to individual actions is the problem that makes reinforcement learning difficult. Surprisingly, there is a solution to this problem. It is based on a field of mathematics called *dynamic programming*, and it involves just two basic principles. First, if an action causes something bad to happen immediately, such as crashing the plane, then the system learns not to do that action in that situation again. So whatever action the system performed one millisecond before the crash, it will avoid doing in the future. But that principle doesn't help for all the earlier actions which didn't lead to immediate disaster.

The second principle is that if all the actions in a certain situation leads to bad results, then that situation should be avoided. So if the system has experienced a certain combination of altitude and airspeed many different times, whereby trying a different action each time, and all actions led to something bad, then it will learn that the situation itself is bad. This is a powerful principle, because the learning system can now learn without crashing. In the future, any time it chooses an action that leads to this particular situation, it will immediately learn that particular action is bad, without having to wait for the crash.

By using these two principles, a learning system can learn to fly a plane, control a robot, or do any number of tasks. It can first learn on a simulator, then fine tune on the actual system. This technique is generally referred to as dynamic programming, and a slightly closer analysis will reveal how dynamic programming can generate the optimal value function.

The Essence of Dynamic Programming

Initially, the approximation of the optimal value function is poor. In other words, the mapping from states to state values is not valid. The primary objective of learning is to find the correct mapping. Once this is completed, the optimal policy can easily be extracted. At this point some notation needs to be introduced : $V^*(\mathbf{x}_t)$ is the optimal value function where $\mathbf{x}_t$ is the state vector; $V(\mathbf{x}_t)$ is the approximation of the value function; γ is a discount factor in the range [0,1] that causes immediate reinforcement to have more importance (weighted more heavily) than future reinforcement. (A more complete discussion of γ is presented in Section 4.)

In general, $V(\mathbf{x}_t)$ will be initialized to random values and will contain no information about the optimal value function $V^*(\mathbf{x}_t)$. This means that the approximation of the optimal value function in a given state is equal to the true value of that state $V^*(\mathbf{x}_t)$ plus some error in the approximation, as expressed in equation (1)

$$V(\mathbf{x}_t) = e(\mathbf{x}_t) + V^*(\mathbf{x}_t) \tag{1}$$

where $e(\mathbf{x}_t)$ is the error in the approximation of the value of the state occupied at time t. Likewise, the approximation of the value of the state reached after performing some action at time t is the true value of the state occupied at time t+1 plus some error in the approximation, as expressed in equation (2).

$$V(\mathbf{x}_{t+1}) = e(\mathbf{x}_{t+1}) + V^*(\mathbf{x}_{t+1}) \tag{2}$$

As stated previously, the value of state $\mathbf{x}_t$ for the optimal policy is the sum of the reinforcements when starting from state $\mathbf{x}_t$ and performing optimal actions until a terminal state is reached. By this definition, a simple relationship exists between the values of successive states, $\mathbf{x}_t$ and $\mathbf{x}_{t+1}$. This relationship is defined by the Bellman equation and is expressed in equation (3). The discount factor γ is used to exponentially decrease the weight of reinforcements received in the future (A discussion of the function of γ in this equation can be found in Section 4).

$$V^*(\mathbf{x}_t) = r(\mathbf{x}_t) + \gamma V^*(\mathbf{x}_{t+1}) \tag{3}$$

The approximation $V(\mathbf{x}_t)$ also has the same relationship, as shown in equation (4). By substituting the right-hand side of equations (1) and (2) into equation (4) we get equation (5) and expanding yields equation (6).

$$V(\mathbf{x}_t) = r(\mathbf{x}_t) + \gamma V(\mathbf{x}_{t+1}) \tag{4}$$

$$e(\mathbf{x}_t) + V^*(\mathbf{x}_t) = r(\mathbf{x}_t) + \gamma\big(e(\mathbf{x}_{t+1}) + V^*(\mathbf{x}_{t+1})\big) \tag{5}$$

$$e(\mathbf{x}_t) + V^*(\mathbf{x}_t) = r(\mathbf{x}_t) + \gamma e(\mathbf{x}_{t+1}) + \gamma V^*(\mathbf{x}_{t+1}) \tag{6}$$

Using equation (3), $V^*(\mathbf{x}_t)$ is subtracted from both sides of equation (6) to reveal the relationship in the errors of successive states. This relationship is expressed in equation (7).

$$e(\mathbf{x}_t) = \gamma e(\mathbf{x}_{t+1}) \tag{7}$$

The significance of this relationship can be seen by using the simple Markov chain shown in Figure 4. In Figure 4 the state labeled T is the terminal state. The true value of this state is known *a priori*. In other words, the error in the approximation of the state labeled T, e(T), is 0 by definition. An analogy might be the state in which a missile ultimately hits or misses its target. The true value of this state is known: +1 for a hit, -1 for a miss. The process of learning is the process of finding an approximation $V(\mathbf{x}_t)$ that makes equations (3) and (7) true for all states $\mathbf{x}_t$. If the approximation error in state 3 is a factor of γ smaller than the error in state T, which is by definition 0, then the approximation error in state 3 must also be 0. If equation (7) is true for all $\mathbf{x}_t$, then the approximation error in each state $\mathbf{x}_t$ is necessarily 0, *ergo* $V(\mathbf{x}_t)=V^*(\mathbf{x}_t)$ for all $\mathbf{x}_t$.

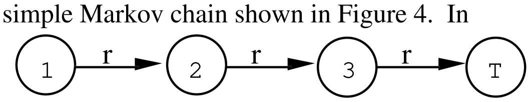


Figure 2

The importance of the discount factor γ can be seen by using another simple Markov chain that has no terminal state (Figure 5). Using a similar argument as that used for Figure 4 one can see that when equation (7) is satisfied the approximation error must be 0 for all states. The only difference in the prior example and in this case is that the error in any given state must be a factor of γ^6 smaller than itself (because there are 6 states in the cycle). Therefore, equation (7) can only be satisfied if the approximation error is 0 in every state.

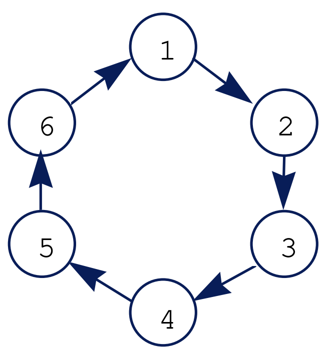


Figure 3

$$e(1) = \gamma e(2)$$
$$e(1) = \gamma^2 e(3)$$
$$e(1) = \gamma^3 e(4)$$
$$e(1) = \gamma^4 e(5)$$
$$e(1) = \gamma^5 e(6)$$
$$e(1) = \gamma^6 \mathrm{e}(1) = 0$$

Therefore, as stated earlier, the process of learning is the process of finding a solution to equation (4) for all states $\mathbf{x}_t$ (which is also a solution to equation (7)). Several learning algorithms have been developed for precisely this task.

Value Iteration

If it is assumed that the function approximator used to represent V^* is a lookup table (each state has a corresponding element in the table whose entry is the approximated state value), then one can find the optimal value function by performing sweeps through state space, updating the value of each state according to equation (8) until a sweep through state space is performed in which there are no changes to state values (the state values have converged).

$$\Delta \mathbf{w}_t = \max_{\mathbf{u}} \left(r(\mathbf{x}_t, \mathbf{u}) + \gamma V(\mathbf{x}_{t+1}) \right) - V(\mathbf{x}_t) \tag{8}$$

In equation (8) $\mathbf{u}$ is the action performed in state $\mathbf{x}_t$ and causes a transition to state $\mathbf{x}_{t+1}$, and $r(\mathbf{x}_t,\mathbf{u})$ is the reinforcement received when performing action $\mathbf{u}$ in state $\mathbf{x}_t$. Figure 6 illustrates the update.

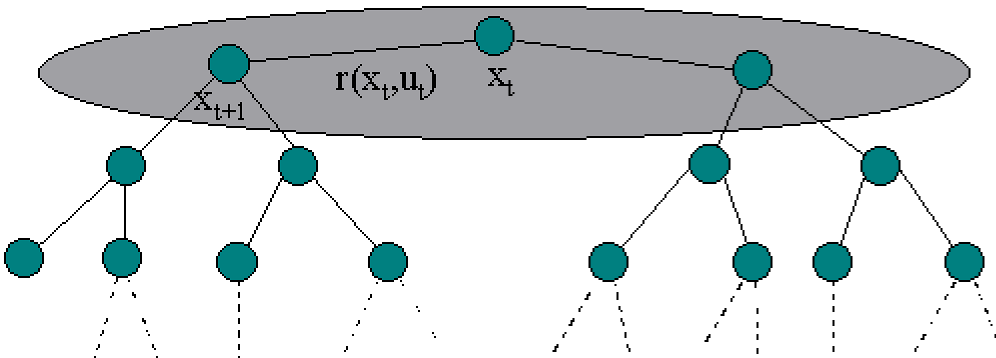


Figure 4

Figure 6 depicts the scope of a single update to the approximation of the value of $\mathbf{x}_t$. Specific to this example, there are two actions possible in state $\mathbf{x}_t$, and each of these actions leads to a different successor state $\mathbf{x}_{t+1}$. In a value iteration update, one must first find the action that returns the maximum value. The only way to accomplish this is to actually perform an action and calculate the sum of the reinforcement received and the (possibly discounted) approximated value of the successor state $V(\mathbf{x}_{t+1})$. This must be done for all actions $\mathbf{u}$ in a given state $\mathbf{x}_t$, and is not possible without a model of the dynamics of the system. For example, in the case of a robot deciding to choose between paths to follow, it is not possible to choose one path, observe the successor state, and then return to the starting state to explore the results of the next available action. Instead, the robot must in simulation perform these actions and observe the results. Then, based on the simulation results, the robot may choose the action that results in the maximum value.

One should note that the right side of equation (8) is simply the difference in the two sides of the Bellman equation defined in equation (4), with the exception that we have generalized the equation to allow for Markov decision processes (multiple actions possible in a given state) rather than Markov chains (single action possible in every state). This expression is the Bellman residual, and is formally defined by equation (9).

$$e(\mathbf{x}_t) = \max_{\mathbf{u}} \left(r(\mathbf{x}_t, \mathbf{u}) + \gamma V(\mathbf{x}_{t+1}) \right) - V(\mathbf{x}_t) \tag{9}$$

$E(\mathbf{x}_t)$ is the error function defined by the Bellman residual over all of state space. Each update (equation (8)) reduces the value of $E(\mathbf{x}_t)$, and in the limit as the number of updates goes to infinity $E(\mathbf{x}_t)=0$. When $E(\mathbf{x}_t)=0$, equation (4) is satisfied and $V(\mathbf{x}_t)=V^*(\mathbf{x}_t)$. Learning is accomplished.

Residual Gradient Algorithms

Thus far it has been assumed our function approximator is a lookup table. The is normally the case in classical dynamic programming. However, this assumption severely limits the size and complexity of the problems solvable. Many real-world problems have extremely large or even continuous state spaces. In practice it is not possible to represent the value function for such problems using a lookup table. Hence, an extension to classical value iteration is to use a function approximator that can generalize and interpolate values of states never before seen. For example, one might use a neural network for the approximation $V(\mathbf{x}_t,\mathbf{w}_t)$ of $V^*(\mathbf{x})$, where $\mathbf{w}_t$ is the parameter vector. The resulting network parameter update is given in equation (10).

$$\Delta \mathbf{w}_t = -\alpha \left[\max_{\mathbf{u}} \left(r(\mathbf{x}_t, \mathbf{u}) + \gamma V(\mathbf{x}_{t+1}, \mathbf{w}_t) \right) - V(\mathbf{x}_t, \mathbf{w}_t) \right] \frac{\partial V(\mathbf{x}_t, \mathbf{w}_t)}{\partial \mathbf{w}_t} \tag{10}$$

It is useful to draw an analogy to the update equation used in supervised learning algorithms when first examining equation (10). In this context, α is the learning rate, $\max_{\mathbf{u}}\left(r\left(\mathbf{x}_t, u\right) + \gamma V\left(\mathbf{x}_{t+1}, \mathbf{w}_t\right)\right)$ is the desired output of the network, $V(x_t,w_t)$ is the actual output of the network, and $\frac{\partial V(\mathbf{x}_t, \mathbf{w}_t)}{\partial \mathbf{w}_t}$ is the gradient of the output of the network with respect to the parameter vector. It "appears" that we are performing updates that will minimize the Bellman residual, but this is not necessarily the case. The "target" value $\max_{\mathbf{u}}\left(r\left(\mathbf{x}_t, \mathbf{u}\right) + \gamma V\left(\mathbf{x}_{t+1}, \mathbf{w}_t\right)\right)$ is a function of the parameter vector **w** at time t. Once the update to **w** is performed, the target has changed because it is now a function of a different parameter vector (the vector at time t+1). It is possible that the Bellman residual has actually been increased rather than decreased. The error function on which gradient descent is being performed changes with every update to the parameter vector. This can result in the values of the network parameter vector oscillating or even growing to infinity.

One solution to this problem is to perform gradient descent on the mean squared Bellman residual. Because this defines an unchanging error function, convergence to a local minimum is guaranteed. This means that we can get the benefit of the generality of neural networks while still guaranteeing convergence. The resulting parameter update is given in equation (11).

$$\Delta \mathbf{w}_t = \alpha\left[r(\mathbf{x}_t) + \gamma V(\mathbf{x}_{t+1}, \mathbf{w}_t) - V(\mathbf{x}_t, \mathbf{w}_t)\right]\left[\frac{\gamma \partial V(\mathbf{x}_{t+1}, \mathbf{w}_t)}{\partial \mathbf{w}_t} - \frac{\partial V(\mathbf{x}_t, \mathbf{w}_t)}{\partial \mathbf{w}_t}\right] \qquad (11)$$

The resulting method is referred to as a *residual gradient* algorithm because gradient descent is performed on the mean squared Bellman residual. Therefore, equation (11) is the update equation for *residual value iteration*, and equation (10) is the update equation for *direct value iteration*. It is important to note that if the MDP is non-deterministic then it becomes necessary to generate independent successor states to guarantee convergence to the correct answer. For a more detailed discussion see Baird (1995); Harmon, Baird, and Klopf (1995); and Harmon and Baird (1996).

Q-Learning

Q-learning (Watkins, 1989 and 1992) is another extension to traditional dynamic programming (value iteration) that solves the following problem.

A *deterministic* Markov decision process is one in which the state transitions are deterministic (an action performed in state $\mathbf{x}_t$ always transitions to the same successor state $\mathbf{x}_{t+1}$). Alternatively, in a *non-deterministic* Markov decision process, a probability distribution function defines a set of potential successor states for a given action in a given state. If the MDP is non-deterministic, then value iteration requires that we find the action that returns the maximum *expected* value (the sum of the reinforcement and the integral over all possible successor states for the given action). For example, to find the expected value of the successor state associated with a given action, one must perform that action an infinite number of times, taking the integral over the values of all possible successor states for that action. The reason this is necessary is demonstrated in Figure 7.

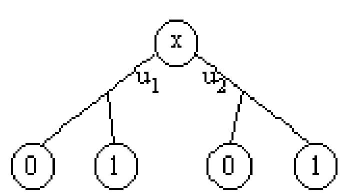


Figure 5

In Figure 7 there are two possible actions in state **x**. Each action returns a reinforcement of 0. Action $\mathbf{u}_1$ causes a transition to one of two possible successor states with equal probability. The same is true for action $\mathbf{u}_2$. The values of the successor states are 0 and 1 for both actions. Value iteration requires that the value of state **x** be equal to the maximum over actions of the sum of reinforcement and the expected value of the successor state. By taking an infinite number of samples of successor states for action $\mathbf{u}_1$, one would be able to calculate that the actual expected value is 0.5. The same is true for action u_2. Therefore, the value of state **x** is 0.5 However, if one were to naively perform value iteration on this MDP by taking a single sample of the successor state associated with each action instead of the integral, then **x** would converge to a value of 0.75. Clearly the wrong answer.

Theoretically, value iteration is possible in the context of non-deterministic MDPs. However, in practice it is computationally impossible to calculate the necessary integrals without added knowledge or some degree of modification. *Q*-learning solves the problem of having to take the max over a set of integrals.

Rather than finding a mapping from states to state values (as in value iteration), *Q*-learning finds a mapping from state/action pairs to values (called *Q*-values). Instead of having an associated value function, *Q*-learning makes use of the *Q*-function. In each state, there is a *Q*-value associated with each action. The definition of a *Q*-value is the sum of the (possibly discounted) reinforcements received when performing the associated action and then following the given policy thereafter. Likewise, the definition of an optimal *Q*-value is the sum of the reinforcements received when performing the associated action and then following the optimal policy thereafter.

In the context of *Q*-learning, the value of a state is defined to be the maximum *Q*-value in the given state. Given this definition it is easy to derive the equivalent of the Bellman equation (equation 4) for *Q*-learning.

$$Q(\mathbf{x}_t, \mathbf{u}_t) = r(\mathbf{x}_t, \mathbf{u}_t) + \gamma \max_{\mathbf{u}_{t+1}} Q(\mathbf{x}_{t+1}, \mathbf{u}_{t+1}) \tag{12}$$

Q-learning differs from value iteration in that it doesn't require that in a given state each action be performed and the expected values of the successor states be calculated. While value iteration performs an update that is analogous to a one level breadth-first search, *Q*-learning takes a single-step sample of a Monte-Carlo roll-out. This process is demonstrated in Figure 8.

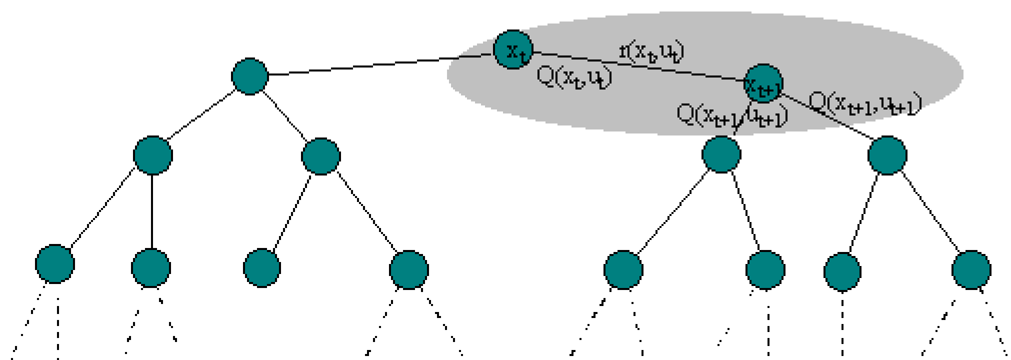


Figure 6

The update equation in Figure 8 is valid when using a lookup table to represent the *Q*-function. The *Q*-value is a prediction of the sum of the reinforcements one will receive when performing the associated action and then following the given policy. To update that prediction $Q(\mathbf{x}_t,\mathbf{u}_t)$ one must perform the associated action $\mathbf{u}_t$, causing a transition to the next state $\mathbf{x}_{t+1}$ and returning a scalar reinforcement $r(\mathbf{x}_t,\mathbf{u}_t)$. Then one need only find the maximum *Q*-value in the new state to have all the necessary information for revising the prediction (*Q*-value) associated with the action just performed. *Q*-learning does not require one to calculate the integral over all possible successor states in the case that the state transitions are non-deterministic. The reason is that a single sample of a successor state for a given action is an *unbiased estimate* of the expected value of the successor state. In other words, after many updates the *Q*-value associated with a particular action will converge to the expected sum of all reinforcements received when performing that action and following the optimal policy thereafter.

Residual Gradient and Direct Q-learning

As it is possible to represent the value function with a neural network in the context of value iteration, so it is possible to represent the Q-function with a neural network in the context of Q-learning. The information presented in the discussion of value iteration concerning convergence to a stable value function is also applicable to guaranteeing convergence to a stable Q-function. Equation (13) is the update equation for direct Q-learning where α is the learning rate, and equation (14) is the update equation for residual gradient Q-learning.

$$\Delta \mathbf{w}_t = -\alpha\left[\left(r(\mathbf{x}_t,\mathbf{u}_t)+\gamma \max_{\mathbf{u}_{t+1}} Q(\mathbf{x}_{t+1},\mathbf{u}_{t+1},\mathbf{w}_t)\right)-Q(\mathbf{x}_t,\mathbf{u}_t,\mathbf{w}_t)\right]\frac{\partial Q(\mathbf{x}_t,\mathbf{u}_t,\mathbf{w}_t)}{\partial \mathbf{w}_t} \tag{13}$$

$$\Delta \mathbf{w}_t = \alpha\left[\left(r(\mathbf{x}_t,\mathbf{u}_t)+\gamma \max_{\mathbf{u}_{t+1}} Q(\mathbf{x}_{t+1},\mathbf{u}_{t+1},\mathbf{w}_t)\right)-Q(\mathbf{x}_t,\mathbf{u}_t,\mathbf{w}_t)\right]\bullet \left[\frac{\partial \gamma Q(\mathbf{x}_{t+1},\mathbf{u}_{t+1})}{\partial \mathbf{w}_t}-\frac{\partial Q(\mathbf{x}_t,\mathbf{u}_t,\mathbf{w}_t)}{\partial \mathbf{w}_t}\right] \tag{14}$$

Advantage Learning

Although Q-learning is a significant improvement over value iteration, it is still limited in scope in at least one important way. The number of training iterations necessary to sufficiently represent the optimal Q-function when using function approximators that generalize scales poorly with the size of the time interval between states. The greater the number of actions per unit time (the smaller the increment in time between actions) the greater the number of training iterations required to adequately represent the optimal Q-function. The explanation for this is demonstrated with a simple example. Figure 9 depicts a Markov decision process with 1000 states. State 0 is the initial state and has a single action available, transition to state 1. State 999 is an absorbing state.

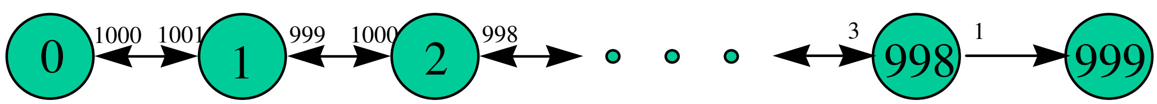


Figure 7

In states 1..998 there are two actions available, transition to either the state immediately to the right or immediately to the left. For example, in state 1, the action of going left will transition to state 0, and the action of going right will transition to state 2. Each transition incurs a cost (reinforcement) of 1. The objective is to minimize the total cost accumulated in transitioning from state to state until the absorbing states is reached. The optimal Q-value for each action is represented by the numbers next to each state. For example, in state 2 the optimal Q-value for the action of going left is 1000, and the optimal Q-value for the action of going right is 998. The optimal policy can easily be found in each state by choosing to perform the action with the minimum Q-value.

When using a function approximator that generalizes over state/action pairs (any function approximator other than a lookup table or equivalent), it is possible to encounter practical limitations in the number of training iterations required to accurately approximate the optimal Q-function. As the time interval between states decreases in size, the required precision in the approximation of the optimal Q-function increases exponentially. For example, the optimal Q-function associated with the MDP in Figure 9 is linear and can be represented by a simple linear function approximator. However, it requires an unreasonably large number of training iterations to achieve the level of precision necessary to generate the optimal policy. The reason for the large number of training iterations is simple. The difference in the Q-values in a given state is small relative to the difference in the Q-values across states (a ratio of approximately 1:1000). For example, the difference in the Q-values in state 1 is 2 (1001-999=2). The difference in the minimum Q-

values in states 1 and 998 is 998 (999-1=998). The approximation of the optimal *Q*-function must achieve a degree of precision such that the tiny differences in *Q*-values in a single state are represented. Because the differences in *Q*-values across states have a greater impact on the mean squared error, during training the network learns to represent these differences first. The differences in the *Q*-values in a given state have only a tiny effect on the mean squared error and therefore get lost in the noise. To represent the differences in *Q*-values in a given state requires much greater precision than to represent the *Q*-values across states. As the ratio of the time interval to the number of states decreases it becomes necessary to approximate the optimal *Q*-function with increasing precision. In the limit, infinite precision is necessary.

Advantage learning does not share the scaling problem of *Q*-learning. Similar to *Q*-learning, advantage learning learns a function of state/action pairs. However, in advantage learning the value associated with each action is called an *advantage*. Therefore, advantage learning finds an advantage function rather than a *Q*-function or value function. The value of a state is defined to be the value of the maximum advantage in that state. For the state/action pair (**x**,**u**) an advantage is defined as the sum of the value of the state and the utility (advantage) of performing action **u** rather than the action currently considered best. For optimal actions this utility is zero, meaning the value of the action is also the value of the state; for sub-optimal actions the utility is negative, representing the degree of sub-optimality relative to the optimal action. The equivalent of the Bellman equation for advantage learning is given in equation (15).

$$A(\mathbf{x}_t,\mathbf{u}_t) = \max_{\mathbf{u}_t} A(\mathbf{x}_t,\mathbf{u}_t) + \frac{\left\langle r(\mathbf{x}_t,\mathbf{u}_t) + \gamma \max\limits_{\mathbf{u}_{t+1}} A(\mathbf{x}_{t+1},\mathbf{u}_{t+1}) \right\rangle - \max\limits_{\mathbf{u}_t} A(\mathbf{x}_t,\mathbf{u}_t)}{\Delta t K} \tag{15}$$

where γ is the discount factor per time step, K is a time unit scaling factor, and <> represents the expected value over all possible results of performing action **u** in state $\mathbf{x}_t$ to receive immediate reinforcement r and to transition to a new state $\mathbf{x}_{t+1}$.

Residual Gradient and Direct Advantage Learning

The number of training iterations required in *Q*-learning scales poorly as the ratio of the time interval between states to the number of states grows small. Advantage learning can find a sufficiently accurate approximation to the advantage function in a number of training iterations that is independent of this ratio. The update equations for direct advantage learning and residual advantage learning are given in equations (16) and (17) respectively. Again, the reader is referred to the subsection in the discussion of value iteration devoted to residual gradient algorithms. For a further discussion of advantage learning see Harmon and Baird (1996).

$$\Delta \mathbf{w}_t = \alpha\left[\left(r(\mathbf{x}_t,\mathbf{u}_t) + \gamma \max_{\mathbf{u}_{t+1}} A(\mathbf{x}_{t+1},\mathbf{u}_{t+1},\mathbf{w}_t)\right)\frac{1}{\Delta t K} + \left(1 - \frac{1}{\Delta t K}\right)\max_{\mathbf{u}_t} A(\mathbf{x}_t,\mathbf{u}_t,\mathbf{w}_t) - A(\mathbf{x}_t,\mathbf{u}_t,\mathbf{w}_t)\right] \bullet \frac{\gamma A(\mathbf{x}_t,\mathbf{u}_t,\mathbf{w}_t)}{\partial \mathbf{w}_t} \tag{16}$$

$$\Delta \mathbf{w}_t = -\alpha \left[\left(r(\mathbf{x}_t,\mathbf{u}_t) + \gamma \max_{\mathbf{u}_{t+1}} A(\mathbf{x}_{t+1},\mathbf{u}_{t+1},\mathbf{w}_t)\right)\frac{1}{\Delta t K} + \left(1-\frac{1}{\Delta t K}\right)\max_{\mathbf{u}_t} A(\mathbf{x}_t,\mathbf{u}_t,\mathbf{w}_t) - A(\mathbf{x}_t,\mathbf{u}_t,\mathbf{w}_t)\right]$$

$$\left[\frac{\gamma\partial \max_{\mathbf{u}_{t+1}} A(\mathbf{x}_{t+1},\mathbf{u}_{t+1},\mathbf{w}_t)}{\partial \mathbf{w}_t}\frac{1}{\Delta t K} + \left(1-\frac{1}{\Delta t K}\right)\frac{\partial \max_{\mathbf{u}_t} A(\mathbf{x}_t,\mathbf{u}_t,\mathbf{w}_t)}{\partial \mathbf{w}_t} - \frac{\gamma A(\mathbf{x}_t,\mathbf{u}_t,\mathbf{w}_t)}{\partial \mathbf{w}_t}\right]$$

(17)

TD(λ)

Consider the Markov chain in Figure 10. The initial state is 0 and the terminal state is 999. Each state transition returns a cost (reinforcement) of 1 and the value of state 999 is defined to be 0. Because this is a Markov chain it is not sensible to suggest that the RL system learn to minimize or maximize reinforcement. Instead, we are concerned exclusively with predicting the total reinforcement received when starting from state n where n is a state in the range [1..998].

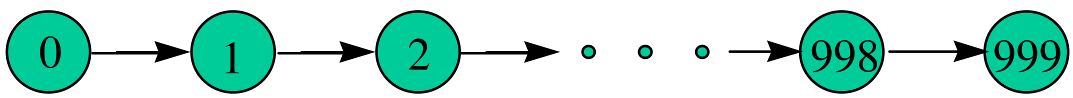


Figure 8

Value iteration, Q-learning, and advantage learning can all solve this problem. However, TD(λ) can solve it faster. In the context of Markov chains, TD(λ) is identical to value iteration with the exception that TD(λ) updates the value of the current state based on a weighted combination of the values of future states, as opposed to using only the value of the immediate successor state. Recall that in value iteration the "target" value of the current state is the sum of the reinforcement and the value of the successor state, in other words, the right side of the Bellman equation (Equation 18).

$$V(\mathbf{x}_t \mathbf{w}_t) = r(\mathbf{x}_t) + \gamma V(\mathbf{x}_{t+1},\mathbf{w}_t) \tag{18}$$

Notice that the "target" is also based on an estimate $V(\mathbf{x}_{t+1},\mathbf{w}_t)$, and this estimate can be based on zero information. Indeed, this is the case much of the time and can be demonstrated using Figure 10. Assume that the value function for this Markov chain is represented using a lookup table. In this case, our lookup table has 1000 elements, each corresponding to a state, and the entry in each element is the value of the corresponding state. Before learning begins entries are initialized to random values. The process of learning starts by updating the value of state 0 to be the sum of the reinforcement received on transition from state 0 to state 1 and the value of state 1. Remember, at this point the value of state 1 is arbitrary. This is true for all states except the terminal state (999) which, by definition, has a value of 0. Because the initial values of states are arbitrary (with the exception of the terminal state), the entire first sweep through the Markov chain (epoch) of training results in the improvement of the approximation of the value function only in state 998. In the first epoch, only in state 998 is the update to the approximation based on something other than an arbitrary value. This is terribly inefficient. In fact, not until 999 epochs of training have been performed will the approximation of the value of state 0 contain any degree of "truth" (the approximation is based on something other than an arbitrary value). In epoch 2 of training, the approximation of the value of state 997 is updated based on an approximation of the value of state 998 that has as its basis the true value of state 999, rather than an arbitrary value. In epoch 3, the approximation of the value of state 996 will be updated based on "truth" rather than an arbitrary value. Each epoch moves "truth" back one step in the chain.

The approximation of the value of state $\mathbf{x}_t$ is updated based on the approximation of the value of the state one step into the future, $\mathbf{x}_{t+1}$. If the value of a state were based on a weighted average of the values of future states, then "truth" would be propagated "back in time" much more efficiently. In our example above, if instead of updating the value of a state based exclusively on the value of the immediate successor state one used the next 2 successor states as the basis of the update, then the number of epochs performed before the value of state 0 is no longer based on an arbitrary value is reduced from 1000 to 500. If the value approximation of state 0 is based on a weighted combination of values of the succeeding 500 states, then only 2 epochs are required before the value approximation of state 0 is based on something other than an arbitrary value.

This is precisely the function of TD(λ) (Sutton, 1988) for $0<\lambda<1$. Instead of updating a value approximation based solely on the approximated value of the immediate successor state, TD(λ) basis the update on an exponential weighting of values of future states. λ is the weighting factor. TD(0), the case of $\lambda=0$, is identical to value iteration for the example problem stated above. TD(1) updates the value approximation of state n based solely on the value of the terminal state.

The parameter update for TD(λ) is given in equation (19).

$$\Delta \mathbf{w}_t = \alpha\Big(r(\mathbf{x}_t) + V(\mathbf{x}_{t+1}, \mathbf{w}_t) - V(\mathbf{x}_t, \mathbf{w}_t)\Big)\sum_{k=1}^{t} \lambda^{t-k} \nabla_{\mathbf{w}} V(\mathbf{x}_k, \mathbf{w}_t) \tag{19}$$

An incremental form of this equation can be derived as follows. Given that $\mathbf{g}_t$ is the value of the sum in (19) for t, we can compute $\mathbf{g}_{t+1}$, using only current information, as

$$\begin{aligned}
\mathbf{g}_{t+1} &= \sum_{k=1}^{t+1} \lambda^{t+1-k} \nabla_{\mathbf{w}} V(\mathbf{x}_k, \mathbf{w}_t) \\
&= \nabla_{\mathbf{w}} V(\mathbf{x}_{k+1}, \mathbf{w}_t) + \sum_{k=1}^{t} \lambda^{t+1-k} \nabla_{\mathbf{w}} V(\mathbf{x}_k, \mathbf{w}_t) \\
&= \nabla_{\mathbf{w}} V(\mathbf{x}_{k+1}, \mathbf{w}_t) + \lambda \mathbf{g}_t
\end{aligned} \tag{20}$$

Notice that equation (19) does not have a max or min term. This suggests that TD(λ) is used exclusively in the context of prediction (Markov chains). One way to extend the use of TD(λ) to the domain of Markov decision processes is to perform updates according to equation (19) while calculating the sum according to equation (20) when following the current policy. When a step of exploration is performed (choosing an action that is not currently considered "best"), the sum of past gradients **g** in equation (20) should be set to 0. The intuition behind this method follows. The value of a state $\mathbf{x}_t$ is defined as the sum of the reinforcements received when starting in $\mathbf{x}_t$ and following the current policy until a terminal state is reached. During training, the current policy is the best approximation to the optimal policy generated thus far. On occasion one must perform actions that don't agree with the current policy so that better approximations to the optimal policy can be realized. However, one might not want the value of the resulting state propagated through the chain of past states. This would corrupt the value approximations for these states by introducing information that is not consistent with the definition of a state value.

One further note. TD(λ) for $\lambda=0$ is equivalent to value iteration. Likewise, the discussion of residual gradient algorithms is applicable to TD(λ) when $\lambda=0$. However, this is not the case for $0<\lambda<1$. No algorithms exist that guarantee convergence for TD(λ) for $0<\lambda<1$ when using a general function approximator.

4 Miscellaneous Issues

Exploration

As stated earlier, the fundamental question in reinforcement learning research is: How do we devise an algorithm that will efficiently find the optimal value function? It was shown that the optimal value function is a solution to the set of equations defined by the Bellman equation (Equation 4). The process of learning was subsequently described as the process of improving an approximation of the optimal value function by incrementally finding a solution to this set of equations. One should notice that the Bellman equation is defined over all of state space. The optimal value function satisfies this equation for ALL $\mathbf{x}_t$ in state space. This requirement introduces the need for *exploration*. Exploration is defined as intentionally choosing to perform an action that is not considered "best" for the express purpose of acquiring knowledge of unseen (or little seen) states. In order to identify a (sub-)optimal approximation, state space must be sufficiently explored.

For example, a robot facing an unknown environment has to spend some time acquiring knowledge of its environment. Alternatively, experience acquired during exploration must also be considered during action selection to minimize the costs (negative reinforcements) associated with learning. Although the robot must explore its environment, it should avoid collisions with obstacles. However, the robot does not know which actions will result in collision until all of state space has been explored. On the other hand, it is possible that a policy that is "sufficiently" good will be recognized without having to explore all of state space. There is a fundamental trade-off between exploration and exploitation (using previously acquired knowledge to direct the choice of action). Therefore, it is important to use exploration techniques that will maximize the knowledge gained during learning while minimizing the costs of exploration and learning time.

For a good introduction to the issues of efficient exploration see Thrun (1992).

Discounted vs. Non-Discounted

The discount factor γ is a number in the range of [0..1] and is used to weight near term reinforcement more heavily than distant future reinforcement. For the purpose of discussion, the update equation for value iteration is shown again as equation (21). The closer γ is to 1 the greater the weight of future reinforcements. The weighting of future reinforcements has a half-life of $\sigma = \log 0.5 / \log \gamma$. For γ=0, the value of a state is based exclusively on the immediate reinforcement received for performing the associated action. For finite horizon Markov decision processes (an MDP that terminates) it is not strictly necessary to use a discount factor. In this case (γ=1), the value of state $\mathbf{x}_t$ is based on the total reinforcement received when starting in state $\mathbf{x}_t$ and following the given policy.

$$\Delta \mathbf{w}_t = \max_{\mathbf{u}} \left(r(\mathbf{x}_t, \mathbf{u}) + \gamma V(\mathbf{x}_{t+1}, \mathbf{w}_t) \right) - V(\mathbf{x}_t, \mathbf{w}_t) \tag{21}$$

In the case of infinite horizon Markov decision processes (an MDP that never terminates), a discount factor is required. Without the use of a discount factor, the sum of the reinforcements received would be infinite for every state. The use of a discount factor limits the maximum value of a state to be on the order of $R/(1-\gamma)$.

5 Conclusion

Reinforcement learning appeals to many researchers because of its generality. Any problem domain that can be cast as a Markov decision process can potentially benefit from this technique. In fact, many researchers view reinforcement learning not as a technique, but rather a particular type of problem that is amenable to solution by the algorithms described above. Reinforcement learning is an extension of classical dynamic programming in that it greatly enlarges the set of problems that can practically be solved. Unlike supervised learning, reinforcement learning systems do not require explicit input-output pairs for training. By combining dynamic programming with neural networks, many are optimistic that classes of problems previously unsolvable will finally be solved.

Acknowledgments

The development of this tutorial was supported under Task 2312R1 by the United States Air Force Office of Scientific Research. We would also like to thank Leemon Baird, Harry Klopf, Eric Blasche, Jim Morgan, and Scott Weaver for useful comments.

6 Glossary

policy - a mapping from states to actions.

reinforcement - a scalar variable that communicates the change in the environment to the reinforcement learning system. For example, if an RL system is a controller for a missile, the reinforcement signal might be the distance between the missile and the target (In which case, the RL system would learn to minimize reinforcement).

Markov decision process - An MDP consists of a set of states X; a set of start states S that is a subset of X; a set of actions A; a reinforcement function R where $R(x,a)$ is the expected immediate reinforcement for taking action a in state x; and an action model P where $P(x'|x,a)$ gives the probability that executing action a in state x will lead to state x'. Note: It is a requirement that the choice of action be dependent solely on the current state observation x. If knowledge of prior actions or states affects the current choice of action then the decision process is not Markov.

deterministic - In the context of states, there is a one-to-one mapping from state/action pairs to successor states. In other words, with probability one, the transition from state x after performing action a will always result in state x'.

non-deterministic - In the context of states, there exists a probability distribution function $P(x'|x,a)$ that gives the probability that executing action a in state x will lead to state x'.

state - The condition of a physical system as specified by a set of appropriate variables.

unbiased estimate - The expected (mean) error in the estimate is zero.

7 References

Baird, L. C. (1995). Residual Algorithms: Reinforcement Learning with Function Approximation. In Armand Prieditis & Stuart Russell, eds. *Machine Learning: Proceedings of the Twelfth International Conference*, 9-12 July, Morgan Kaufmann Publishers, San Francisco, CA.

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

Bertsekas, D. P. (1995). *Dynamic Programming and Optimal Control*. Athena Scientific, Belmont, MA.

Harmon, M. E., Baird, L. C., and Klopf, A. H. (1995). Reinforcement learning applied to a differential game. *Adaptive Behavior*, MIT Press, (4)1, pp. 3-28.

Harmon, M. E., Baird, L. C., and Klopf, A. H. (1994). Advantage Updating Applied to a Differential Game. In Tesauro, Touretzky & Leen, eds. *Advances in Neural Information Processing Systems: Proceedings of the 1994 Conference*, MIT Press, Cambridge, Massachusetts.

Harmon, M. E., and Baird, L. C. (1996). Multi-player residual advantage learning with general function approximation. (Technical Report WL-TR-96-1065). Wright-Patterson Air Force Base Ohio: Wright Laboratory. (available from the Defense Technical Information Center, Cameron Station, Alexandria, VA 22304-6145).

Kaelbling, L. P., Littman, M. L., and Moore, A.W. (1996). Reinforcement Learning: A Survey, *Journal of Artificial Intelligence Research* Volume 4, pp. 237-285.

Littman M. L. (1994). Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the Eleventh International Conference on Machine Learning*, , San Francisco, CA., Morgan Kaufmann, pp. 157-163

Sutton, R. S. (1988). Learning to Predict by the Methods of Temporal Differences. *Machine Learning* 3: 9-44.

Thrun, S. (1992). *The Role of Exploration in Learning Control*. In Handbook for Intelligent Control: Neural, Fuzzy and Adaptive Approaches, Van Nostrand Reinhold, Florence, Kentucky 41022.

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

Watkins, J. C. H., Dayan, P. (1992). Technical Note: *Q*-Learning. *Machine Learning* 8: 279-292.

8 Bibliography

Applications of Reinforcement Learning

Tesauro, G.J. (1995). Temporal Differences Learning and TD-Gammon. *Communications of the ACM*, 38:58-68.

Boyan, J. A., and Littman, M. L. (1994). Packet routing in dynamically changing networks: A reinforcement learning approach. In Cowan, J. D., Tesauro, G., and Alspector, J. (eds.), Advances in Neural Information Processing Systems: Proceedings of the 1994 Conference, San Francisco, CA: Morgan Kaufmann.

Crites, R. H., and Barto, A.G. (1996). Improving Elevator Performance Using Reinforcement Learning.. In Touretzky, Mozer & Hasselmo, eds. *Advances in Neural Information Processing Systems: Proceedings of the 1995 Conference*, MIT Press, Cambridge, Massachusetts.

Singh, S., and Bertsekas, D. P. (1996). Reinforcement Learning for Dynamic Channel Allocation in Cellular Telephone Systems. To appear *in Advances in Neural Information Processing Systems: Proceedings of the 1996 Conference*, MIT Press, Cambridge, Massachusetts.

Zhang, W., and Dietterich, T.G. (1996). High-Performance Job-Shop Scheduling With a Time-Delay TD(λ) Network. In Touretzky, Mozer & Hasselmo, eds. *Advances in Neural Information Processing Systems: Proceedings of the 1995 Conference*, MIT Press, Cambridge, Massachusetts.