Reinforcement Learning

Andrew G. Barto

Department of Computer Science

University of Massachusetts, Amherst MA 01003

Running Head: Reinforcement Learning

Correspondence:

Andrew G. Barto

Deptartment of Computer Science

140 Governors Drive

University of Massachusetts

Amherst, MA 01003-4610

Phone: 413 545-2109

Fax: 413 545-1249

email: barto@cs.umass.edu

Introduction

The term reinforcement comes from studies of animal learning in experimental psychology, where it refers to the occurrence of an event, in the proper relation to a response, that tends to increase the probability that the response will occur again in the same situation. Although not used by psychologists, the term "reinforcement learning" has been widely adopted by theorists in artificial intelligence and engineering to refer to learning tasks and algorithms based on this principle of reinforcement. The simplest reinforcement learning methods use the commonsense idea that if an action is followed by a satisfactory state of affairs, or an improvement in the state of affairs, then the tendency to produce that action is strengthened, i.e., reinforced. The ideas of reinforcement learning have been present in engineering for many decades (e.g., Mendel and McClaren 1970) and in artificial intelligence since its earliest days (Minsky 1954, 1961; Samuel 1959; Turing 1950). It is only relatively recently, however, that development and application of reinforcement learning methods have occupied a significant number of researchers in these fields. Fueling this interest are two basic challenges: 1) designing autonomous robotic agents that can operate under uncertainty in complex dynamic environments, and 2) finding useful approximate solutions to very large-scale dynamic decision-making problems.

Reinforcement learning is usually formulated as an *optimization problem* with the objective of finding a strategy for producing actions that is optimal, or best, in some well-defined way. In practice, however, it is usually more important for a reinforcement learning system continue to improve than it is for it to actually achieve optimal behavior. Reinforcement learning differs from the more commonly studied paradigm of supervised learning, or "learning with a teacher," in significant ways that we discuss in the course of this article. It also differs significantly from various forms of unsupervised learning. The

article REINFORCEMENT LEARNINING IN MOTOR CONTROL contains additional information. For a more detailed introductory treatment, the reader should consult Sutton and Barto (1998); for a more in-depth mathematical treatment, the reader should consult Bertsekas and Tsitsiklis (1996).

The Reinforcement Learning Problem

Think of an agent interacting with its environment over a potentially infinite sequence of discrete time steps $t = 1, 2, 3, \ldots$. At each time step t, the reinforcement learning agent receives some representation of the environment's current *state*, $s_t \in S$, where S is the set of possible states, and on that basis executes an *action*, $a_t \in A(s_t)$, where $A(s_t)$ is the set of actions that can be executed in state s_t. One time step later, the agent receives a *reward*, r_{t+1}, a real number, and finds itself facing a new state, $s_{t+1} \in S$ (Fig. 1). The reward and new state are not only influenced by the agent's action, they are also influenced by the state, s_t, in which the action was taken, and they can depend on random factors as well. Throughout this article we assume that S and $A(s)$, $s \in S$, are finite sets, but extension to infinite sets is possible, as is extension to continuous-time formulations.

The rule the agent uses to select actions is called its *policy*. It is a function, often denoted π, that for each state assigns a probabilty to each possible action: for all $s \in S$ and all $a \in A(s)$, $\pi(s, a)$ is the probability that the agent executes a when in state s. While intereacting with its environment, a reinforcement learning agent adjusts its policy based on its accumulating experience to try to improve the the amount of reward it receives over time. More specifically, it tries to maximize the *return* it receives after each time step. The most commonly-studied type of return is the *discounted return*. If

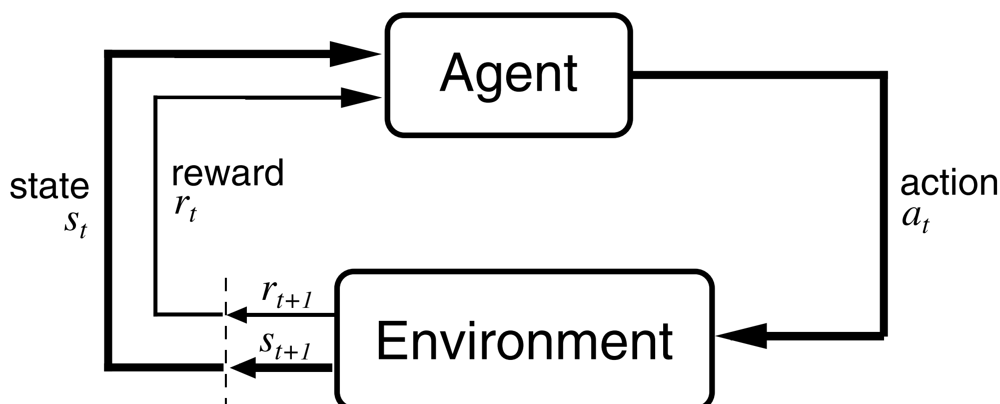

Figure 1: A Reinforcement Learning Model. A reinforcement learning agent and its environment interact over a sequence of discrete time steps. The *actions* are the choices made by the agent; the *states* provide the agent's basis for making the choices; and the *rewards* are the basis for evaluating these choices.

$r_{t+1}, r_{t+2}, r_{t+3}, \ldots$, denotes the sequence of rewards received after time step t, then the discounted return for step t is

$$\sum_{k=1}^{\infty} \gamma^{k-1} r_{t+k}, \tag{1}$$

where $\gamma \in [0, 1)$ is the *discount factor*. A reinforcement learning agent adjusts its policy to try maximize the expected value of quantity for all $t \geq 0$.

The discount factor determines the present value of future rewards. If $\gamma = 0$, the agent is only concerned with maximizing immediate rewards: its objective would be to learn how to act at each time step t so as maximize only r_{t+1}. But in general, acting to maximize immediate reward can reduce access to future rewards so that a longer-term return may actually be reduced. As γ approaches one, the objective takes future rewards into account more strongly: the agent becomes more far-sighted. Discounting is used because it is mathematically the simplest way to deal with cases in which the agent and

environment can interact for an unbounded number of time steps. In many problems only finite numbers of steps can ever happen in each learning trial so that γ can be set to one. These are called *episodic* problems. Other definitions of return have been extensively studied as well.

This model of the reinforcement learning problem is based on the theory of *Markov decision processes* (MDPs), which has been extensively developed in decision theory and stochastic control (see, e.g., Bertsekas 1987). An MDP has the property that the environment satisfies the Markov property, which means that environment state at any time step $t > 0$ provides the same information about what will happen next as would the entire history of the process up to step t. A full specification of an MDP includes the probabilistic details of how state transitions and rewards are influenced by states and actions, i.e., a full probabilistic model of the environment and how it is influenced by the agent's actions. The objective is to compute an *optimal policy*, i.e., a policy that maximizes the expected return from each state. In theory, this can be done using any of several stochastic dynamic programming algorithms, although their computational complexity makes them impractical for large-scale problems.

Reinforcement learning has much in common with this traditional study of MDPs, but it emphasizes approximating optimal behavior during on-line behavior instead of computating optimal policies off-line on the basis of known probabilistic models. In particular, the objective in reinforcement learning is actually not to compute an optimal policy; it is instead to allow the agent to receive as much reward as possible during its behavior. This does not always require a policy that is optimal for all possible states since the agent may not visit all of these states while it is behaving.

Following are some key observations about the reinforcement learning problem:

1. *Uncertainty* plays a central role in reinforcement learning. The agent's environment and its own behavior can be subject to random fluctuations so that the outcomes of decisions cannot be known beforehand with complete certainty. An accurate probabilistic model of the these uncertainties may, or may not, be available to the agent.

2. The reward input to the agent can be any scalar signal evaluating the agent's behavior. It might indicate just success when a goal state is reached, just failure while not reaching a goal state, or it might provide moment-by-moment evaluatuations of on-going behavior (as, for example, in giving the amount of energy currently being consumed while a task is being accomplished). Moreover, multiple evaluation criteria can be combined in various ways to form the scalar reward signal (for example, via a weighted sum).

3. An important difficulty faced by a reinforcement learning system is the *credit-assignment problem* (Minsky 1961): How do you distribute credit for success among the many decisions that may have been involved in producing it? (See also REINFORCEMENT LEARNING IN MOTOR CONTROL.)

4. A reinforcement learning system often has to forgo immediate reward in order to obtain more reward later or over the long run. This kind of "sacrificing" behavior arises because the agent's actions influence not only each reward input but also the environment's state transitions. An action may be preferred because it sets the stage for a large reward later rather for its immediate reward.

5. The reward signal does not directly tell the agent what action is best; it only evaluates the action taken. A reward input also does not directly tell the agent

how to change its actions. These are key features distinguishing reinforcement learning from supervised learning, and we discuss them further below.

6. Reinforcement learning algorithms are *selectional* processes. There must be *variety* in the action-generation process so that the consequences of alternative actions can be compared to select the best. Behavioral variety is called *exploration*; it is often generated through randomness, but it need not be.

7. Reinforcement learning involves a conflict between *exploitation* and *exploration.* In deciding which action to take, the agent has to balance two conflicting objectives: it has to exploit what it has already learned to obtain high rewards, and it has to behave in new ways—explore—to learn more. Because these needs ordinarily conflict, reinforcement learning systems have to somehow balance them. In control engineering, this is known as the conflict between control and identification.

8. Some researchers think of reinforcement learning as a form of supervised learning (because the reward input is a kind of supervision), and others think of it as a form of unsupervised learning (because the reward input is not like the label of an example). There is some truth to each of these views, but reinforcement learning is really different from both. A key distinguishing feature is the presence in reinforcement learning of the conflict between exploitation and exploration. This is absent from supervised and unsupervised learning unless the learning system is also engaged in influencing which training examples it sees.

Value Functions

The most commonly studied reinforcement learning algorithms are based on estimating *value functions*, which are scalar functions of states, or of state-action pairs, that tell how good it is for the agent to be in a state, or to take an action in a state. The notion of "how good" is the return expected to accumulate over the future, which is well-defined if the Markov property holds and the agent's policy is specified.

If the agent uses policy π, then the state value function V^π gives the *value*, $V^\pi(s)$, of each $s \in S$, which is the return expected to accumulate over the time period after visiting s, assuming that actions are chosen according to π. For the discounted return defined by Eq. **??**, the value of state s is

$$V^\pi(s) = \mathrm{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s], \tag{2}$$

where E_π is the expected value given that policy π is followed. A state's *optimal value*, $V^*(s)$, is the return expected after visiting s assuming that actions are chosen optimally, i.e., it is the largest expected return possible after visiting s.

Similarly, the *action value* of taking action a in state s under a policy π, denoted $Q^\pi(s, a)$, is the expected return starting from s, taking the action a, and thereafter following policy π:

$$Q^\pi(s, a) = \mathrm{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a]. \tag{3}$$

The *optimal action value* of taking action a in state s, denoted $Q^*(s, a)$, is the expected return starting from s, taking the action a, and thereafter following an optimal policy.

Value functions are useful because of several properties of MDPs. If V^* is known, optimal policies can be found by looking ahead only one time step. That is, if s_t is the

state at step t, then an optimal action is any $a \in A(s_t)$ that maximizes the expected value of $r_{t+1} + \gamma V^*(s_{t+1})$. Thus, given V^* and an accurate model of the immediate effects on the environment of all of the actions, acting optimally does not require deep lookahead because V^* summarizes the effects of future behavior. If Q^* is known, then finding optimal actions is even easier. An optimal action at step t is any action that maximizes $Q^*(s_t, a)$. In this case, it is not necessary to look ahead one step, so that no model is needed of the effect of actions on the environment. This is what makes reinforcement learning algorithms that use action-value functions a popular choice in many applications. Any such one-step ahead maximizing action for a state value function, or a maximizing action for an action-value function, is called a *greedy* action with respect to that function.

Value functions that depend on a policy, that is, V^π and Q^π, are useful for improving behavior because of the *policy improvement property*. Suppose the agent is deciding which action to execute in a state. It could pick an action using its current policy, π, or it could select some other action. If it picks an action that is greedy with respect to V^π, and otherwise follows π, then its performance is guaranteed to be at least as good as it would have been under π, and possibly better. This fact is the basis of the policy improvement, or policy iteration, dynamic programming algorithm, and it motivates many reinforcement learning algorithms as we explain below.

A fundamental property of value functions is that they satisfy particular consistency conditions if the Markov property holds. For any policy π and any state s the following is true (for the discounted return case):

$$
\begin{aligned}
V^\pi(s) &= \mathrm{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s] \\
&= \mathrm{E}_\pi[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s]
\end{aligned}
$$

$$= \quad E_\pi[r_{t+1} + \gamma V^\pi(s_{t+1})|s_t = s]. \tag{4}$$

An analogous consistency condition holds for values of Q^π. Similarly, V^* satisfies the following equation for all $s \in S$:

$$V^*(s) = \max_a E[r_{t+1} + \gamma V^*(s_{t+1})|s_t = s, a_t = a], \tag{5}$$

and Q^* satisfies

$$Q^*(s,a) = E[r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a')|s_t = s, a_t = a], \tag{6}$$

for all pairs (s, a), $s \in S$, $a \in A(s)$.

If a model is available giving the probabilistic details of how the environment responds to actions, then these equations (or more precisely, these *sets* ofequations) are completely specified and can in pr inciple be solved using one of a variety of methods for solving systems of linear equations (to obtain V^π or Q^π) or nonlinear equations (to obtain V^* or Q^*). These are often called *Bellman Equations*, after Richard Bellman who introduced the term dynamic programming to refer to a collection of solution methods (Bellman 1957). There are many books that explain dynamic programming, e.g., Bertsekas, 1987.

Solving Bellman equations is therefore one route to finding optimal policies. Unfortunately, in many problems of interest one does not have the complete Markov model of the environment needed to fully define the Bellman equations, or the state set may be so large that it is not computationally feasible to exactly solve the Bellman equations. Unless some special additional structure can be exploited, one has to settle for approximate solutions.

Reinforcement Learning based on Value Functions

Value functions are used in several different ways in reinforcement learning. One approach uses the *actor-critic architecture*, which maintains a representation of both a value function and a policy (Fig. 2). To select actions, an agent using this architecture consults its current policy, represented by the *actor* component. The policy might be represented by a lookup table, by an artificial neural network with its input coding the current state and its output coding the action to be taken, or by some other means. To evaluate the action just taken, the *critic* component is consulted, which maintains an estimate of the value function of the current policy. The action is considered to be "good" ("bad") to the extent that it leads to a next state with a value higher (lower) than that of s, both state values being estimated by the critic. Upon receiving this evaluation, the actor updates the policy by making a good action more likely to be selected upon revisiting s, or a bad action less likely, thus implementing a version of Edward Thorndike's famous "Law of Effect" (Thorndike 1911). The critic component then updates its value function estimate using a temporal difference learning algorithm of the kind described below.

Barto, Sutton, and Anderson (1983) used this architecture for learning to balance a simulated pole mounted on a cart. Their perspective was that the critic provides an internal reinforcement signal—changes in estimated values—that provides *immediate* action evaluations, even though the goal is to maximize reward over the long-term. To the extent that the critic's value estimates are correct given the actor's current policy, the actor actually learns to increase the total amount of future reinforcement. This method thus relies on the policy improvement property. Although not a failsafe approach from a theoretical perspective, it is often successful in improving the agent's behavior.

Another type of reinforcement learning algorithm that uses value functions main-

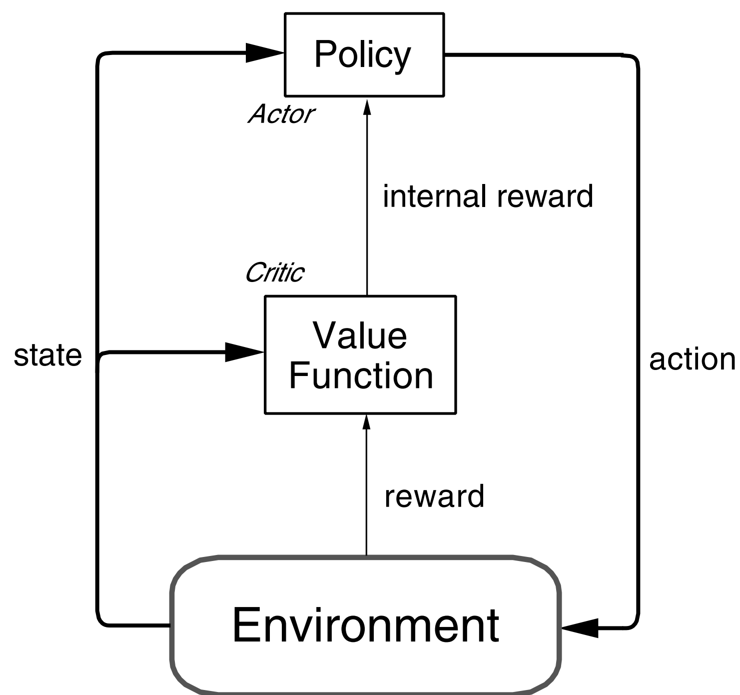

Figure 2: Actor-Critic Architecture. The *critic* provides an internal reward signal to an *actor* which learns a policy for interacting with the environment.

tains an estimate of the current policy's value function but does not keep an explicit representation of the current policy. Instead, it selects actions solely by consulting its current value function estimate. At each time step, the agent selects an action that is either greedy with respect to its current estimate of the value function, or is an exploratory action chosen on some other basis (see below). If state values are being estimated, finding a greedy action requires projecting ahead one step using an environment model; if action values are being estimated, no look-ahead is required, as explained above. Like actor-critic methods, this approach also relies on the policy improvement property, but since there is no separate policy representation and no separate policy update rule, it is more closely related to various dynamic programming algorithms and is therefore somewhat easier to understand.

Estimating Value Functions

The simplest method for estimating the value function of the current policy while the agent is behaving is to average an ensemble of returns actually observed. For example, if an agent follows policy π and maintains, for each state s encountered, an average of the actual returns that have followed that state, then the averages will converge to $V^\pi(s)$ as the number of times that state is encountered approaches infinity. If separate averages are kept for each action, a, taken in a state, then these averages will similarly converge to the action values, $Q^\pi(s, a)$. This is easiest to do in episodic problems where return is accumulated over finite numbers of time steps. Methods like this are sometimes called *simple Monte Carlo* value estimation methods.

Another class of value estimation methods are called *temporal difference* (TD) algorithms (Sutton 1988). The most basic TD algorithm, called *tabular TD(0)*, estimates

V^π while the agent is behaving according to π and is applicable when the state set is small enough to store the state values in a lookup table with a separate entry for the value of each state. Suppose the agent is in state s, executes action a, and then observes the resulting reward r and the next state s'. TD(0) updates the current estimate of the value of state s, $V(s)$, using the following update:

$$V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)], \tag{7}$$

where α is a positive step-size parameter. TD algorithms are based on the consistency condition expressed by the Bellman equations. This TD algorithm is designed to move the term $r + \gamma V(s') - V(s)$, called the *TD error*, toward zero for every state. If the expected TD error could be made to equal zero for every state, then the corresponding Bellman equation (Eq. **??**) would be satisfied. An update of this general form is often called a *backup* because the value of a state is moved toward the current value of a successor state, plus any reward that is received on the transition.

This algorithm converges to the correct state values under certain conditions (Sutton 1988). This and other TD algorithms have been extended to include *eligibility traces*, which allow values to be backed up over more than one time step. When so extended, these are called TD(λ) algorithms, where λ is a parameter determining the temporal characteristics of the backups: λ ranges from 0 (no eligibility traces as above) to 1 (resulting in a simple Monte Carlo method). Forms of this TD algorithm are also known as *adaptive critic algorithms*.

Another TD algorithm, known as *Q-learning*, was proposed by Watkins in 1989 (see Sutton and Barto 1998). This algorithm directly estimates Q^* without relying on the policy improvement property. Its tabular form works as follows. Suppose the agent is in state s, executes action a, and then observes the resulting reward r and the next state s'.

The Q-learning algorithm updates the action value estimate, $Q(s, a)$, of the pair (s, a) using the following backup:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)], \quad (8)$$

where α is a positive step-size parameter. If α decreases appropriately with time and each state-action pair would be visited infinitely often in the limit, then this algorithm converges to $Q^*(s, a)$ for all $s \in S$ and $a \in A(s)$ with probability one. Unless it is known that the environment is deterministic, the "infinitely often" requirement is necessary for this kind of strong convergence of any method that is based, as this one is, on sampling environment state transitions and rewards. Letting the agent sometimes select actions randomly from a uniform distribution is one simple way to help the agent maintain enough variety in its behavior in order to try to satisfy this condition. Otherwise, the agent executes actions that are greedy with respect to its current estimate of Q^*.

Closely related to Q-learning is the *Sarsa* algorithm. Suppose the agent is in state s, executes action a, observes reward r and the next state s', and then executes action a'. Then the Sarsa update is

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)], \quad (9)$$

which is the same as the Q-learning update (Eq. **??**) except that instead of taking the maximum over the actions available in s', it uses the action, a', which was actually executed. (This requirement of s, a, r, s', a' is what accounts for the algorithm's name.) Notice that if actions are always greedy with respect to the current estimate of Q^*, then Sarsa is the same as Q-learning. Despite this similarity, Sarsa and Q-learning have somewhat different properties (see Sutton and Barto 1998). Whereas Q-Learning converges to Q^* independently of the agent's behavior (as long as the conditions for

convergence are satisfied), Sarsa converges to an action-value function that is optimal given the agent's mode of exploration. Like the TD algorithm for state values described above, both Q-learning and Sarsa can be extended to include eligibility traces.

TD algorithms are closely related to dynamic programming algorithms, which also use backup operations derived from Bellman equations. There are two main differences. First, a dynamic programming backup computes the expected value of successor states using the state-transition distribution of the MDP, whereas a TD backup uses a sample from this distribution. (TD backups are sometimes called *sample backups*, in contrast to the *full backups* of dynamic programming.) A second difference is that dymanic programming uses multiple exhaustive "sweeps" of the MDP's state set, whereas TD algorithms operate on states as they occur in actual or simulated experiences. These differences make it possible to use TD algorithms on problems for which it is not feasible to use dynamic programming.

Function approximation

Instead of storing the estimated values of states or state-action pairs in lookup tables, it is possible to represent them more compactly. This is an important feature of reinforcement learning because it enables its use for problems whose state sets are too large to allow explicit representation of each value estimate, and hence too large for text-book dynamic programming algorithms to be feasible. Very large state sets often arise due to combinatorial explosions in representing states that are configurations of discrete objects. They also arise when multi-dimensional continuous spaces are discretized (prompting Bellman to coin the familiar phrase "curse of dimensionality"). For example, the game of Backgammon, to which reinforcement learning has been applied with striking success

(Tesauro 1992), has more than 10^{20} states.

Any of the TD backup rules described above can be used to derive an update rule for a parameterized function approximation method of the type developed for supervised learning. Many reinforcement learning applications have used multi-layer artificial neural networks and error backpropagation (see BACKPROPAGATION). To do this requires representing states or state-action pairs as feature vectors. Training examples are extracted from the agent's behavioral trajectory. For example, suppose one approximated the value of any state s by a function of a feature vector $\vec{\phi}(s)$ and parameter vector $\vec{\theta}$: $V(s) = f(\vec{\phi}(s), \vec{\theta})$. Then the agent's experience of observing state s, followed by reward r and successor state s', would yield the training example consisting of input vector $\vec{\phi}(s)$ and the target output $r + \gamma f(\vec{\phi}(s'), \vec{\theta}_t)$. A gradient-descent update of $\vec{\theta}$ derived from Eq. **??** is

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha[r + \gamma f(\vec{\phi}(s'), \vec{\theta}_t) - f(\vec{\phi}(s), \vec{\theta}_t)]\nabla_{\vec{\theta}_t} f(\vec{\phi}(s), \vec{\theta}_t).$$

Notice that unlike the case in supervised learning, the target output also depends on the parameter vector. This complicates the behavior and analysis of this type of learning rule. Convergence results have been derived for TD(λ) algorithms in the case of function approximators that are linear in $\vec{\theta}$, and counterexamples to convergence have been presented for Q-learning. The reader should consult Sutton and Barto (1998) and Bertsekas and Tsitsiklis (1996) for discussions of these results. Despite a shortage of theoretical guarantees, many reinforcement learning systems using nonlinear function approximators have demonstrated good performance, and much current research is examining these issues.

Exploration

Reinforcement learning agents have to explore: they have to sometimes select actions that appear to be suboptimal according to their current state of knowledge (e.g., current value and/or policy estimates). Otherwise, behavior can become irretrievably suboptimal as the knowledge base comes to reflect only limited experiences. Balancing exploration with the exploitation of current knowledge is a subtle problem that has been extensively studied. In principle it is possible to optimally balance exploration and exploitation by solving an MDP whose states are *belief states* that summarize the agent's entire history of observations and actions. But this approach is not feasible for most tasks of interest.

Several simple heuristic exploration methods are usually used in applications of reinforcement learning. In the simplest, the agent selects ϵ-*greedy* actions. This means that with probability $1-\epsilon$, the agent exploits its current knowledge by selecting a greedy action, that is, an action that is optimal given its current value estimates, and with probability ϵ, it selects an action at random, uniformly, independently of its of its current value estimates. Somewhat more complicated is the *softmax* method which selects actions according to a Boltzmann distribution based on the current action values. This gives actions with higher estimated values higher probabilities of being selected, with a "temperature" parameter determining how much an action's estimated value influences its selection probability. More sophisticated methods monitor the degree of certainty involved in action choices and direct exploration accordingly. How to design methods for balancing exploration and exploitation that are practical, effective, and amenable to theoretical treatment is an important research area.

Direct Policy Search

Not all reinforcement learning methods use value functions. It is possible to search directly in the space of policies. For example, the amount of reward that a policy yields can be estimated by running the policy for some number of time steps, possibly repeating many times from different initial states. This provides an evaluation of the entire policy that can be used to direct search in policy space. The success of the approach usually depends on suitably parameterizing policies by vectors of real numbers so that the search can be conducted in parameter space using any of a large number of optimization algorithms. Some of these algorithms require estimates of the gradient of the policy evaluation with respect to the parameters, which can be also be extracted from sample policy executions.

If the agent-environment interaction is approximately Markov, TD can methods take advantage of local consistency conditions to obtain state-localized information about how to improve a policy. On the other hand, direct policy search does not depend on the Markov property and so can be used when state information is not close to being available. Direct policy search methods also do not require the use of function approximation methods to represent value functions. Offsetting these advantages of direct policysearch methods, however, is the more coarse form of credit assignment that is possible and the difficulty of efficiently evaluating entire policies. Which type of method is to be recommended is highly problem dependent. The actor-critic architecture can be considered to combine aspects of value function and direct policy search algorithms, and there is considerable interest in this hybrid approach.

Using Environment Models

Algorithms like Q-learning and Sarsa do not need a model of the agent's environment. They can learn from the agent's actual experience as the agent behaves in the real world. However, many reinforcement learning systems do take advantage of environment models. For example, algorithms like Q-learning and Sarsa are often applied to experience generated as the agent interacts with a simulation of its environment. This not anly allows much faster learning (since simulations can run much faster than real time), it eliminates the potential of catastrophic consequences that can occur in some domains when a learning system is given control over a real system.

Sutton and Barto (1998) called models that can support learning from simulations *sample models*. In contrast, stochastic dynamic programming algorithms need *distribution models* which explicitly represent the environment's state-transition and reward probabilities. Since sample models can sometimes be much easier to construct than distribution models, their ability to form policies through simulation is an important advantage of reinforcement learning methods for some applications. It is also easy to devise algorithms that learn from both real and simulated experience. Other reinforcement learning algorithms take advantage of distribution models by using full, instead of sample, backups, while still applying backups to states encountered along simulated or actual behavioral trajectories. This approach makes each backup more informative than a sample backup but avoids the exhaustive sweeping of dynamic programming.

Determining a policy from an environment model—either a distribution or a sample model—is a form of *planning*. Reinforcement learning algorithms that use models are not clearly distinct from some types of planning algorithms. Their main distinguishing characteristic is probably that they often do not fully complete a planning process before

committing to actions. The planning process is extended over time, with knowledge in the form of a value function and/or a policy accumulating as behavior continues. Model-based reinforcement learning is closely related to *decision theoretic planning* in artificial intelligence, which also makes use of the MDP formalism.

Elaborations and Extensions

Among the many topics being addressed by current reinforcement learning research are: extending theoretical results to include parameterized function approximation methods; understanding how exploratory behavior is best introduced and controlled; learning under conditions in which the environment state cannot be fully observed (related to the theory of Partially Observable MDPs, or POMDPs); exploiting structure present when states and/or actions are represented as vectors giving the values of descriptive variables (formalised in terms of *factored* or *structured* MDPs); introducing various forms of abstraction such as temporally-extended actions and hierarchy (which rely strongly on the theory of Semi-Markov Decision Processes, or SMDPs). Finally, researchers are studying the relationship of computational reinforcement learning theories to brain reward mechanisms. Strong parallels exsit between TD learning and the activity of dopamine neurons (Schultz 1998; DOPAMINE, ROLES OF).

REFERENCES

Barto, A. G., R. S. Sutton, and C.W. Anderson, 1983. Neuronlike elements that can solve difficult learning control problems, *IEEE Transactions on Systems, Man and Cybernetics*, 13:835-846. Reprinted in *Neurocomputing: Foundations of Research*, (J. A. Anderson and E. Rosenfeld, eds.), Cambridge, MA: MIT Press, 1988, pp. 535-549.

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

* Bertsekas, D. P. 1987. *Dynamic Programming: Deterministic and Stochastic Models.* Englewood Cliffs, NJ: Prentice-Hall.

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

Mendel, J. M. and R. W. McLaren, 1970. Reinforcement learning control and pattern recognition systems, in *Adaptive, Learning and Pattern Recognition Systems: Theory and Applications*, (J. M. Mendel and K. S. Fu, eds.), New York: Academic Press, pp. 287-318.

Minsky, M. L. 1954. *Theory of Neural-Analog Reinforcement Systems and its Application to the Brain-Model Problem*, Ph.D. thesis, Princeton University.

Minsky, M. L. 1961. Steps toward artificial intelligence. *Proceedings of the Institute of Radio Engineers*, 49:8-30. Reprinted in *Computers and Thought*, E. A. Feigenbaum and J. Feldman, eds.), New York: McGraw-Hill, 1963, pp. 406-450.

Samuel, A. L. 1959. Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development,* 210-229. Reprinted in *Computers and Thought*, E. A. Feigenbaum and J. Feldman, eds.), New York: McGraw-Hill, 1963, pp.71-105.

Schultz, W. 1998. Predictive reward signal of dopamine neurons. *Journal of Neurophysiology*, 80:1-27.

Sutton, R. S. 1988. Learning to predict by the method of temporal differences. *Machine Learning*, 3:9-44.

* Sutton, R. S., and A. G. Barto, 1998. *Reinforcement Learning: An Introduction.* Cambridge, MA: MIT Press.

Tesauro, G. J. 1992. Practical issues in temporal difference learning. *Machine Learning*, 8:257-277.

Thorndike, E. L. 1911. *Animal Intelligence*, Darien, CT: Hafner.

Turing, A. M. 1950. Computing machinery and intelligence. *Mind*, 59:433-460. Reprinted in *Computers and Thought*, E. A. Feigenbaum and J. Feldman, eds.), New York: McGraw-Hill, 1963, pp. 11-35.

FIGURE CAPTIONS

Figure 1. A Reinforcement Learning Model. A reinforcement learning agent and its environment interact over a sequence of discrete time steps. The *actions* are the choices made by the agent; the *states* provide the agent's basis for making the choices; and the *rewards* are the basis for evaluating these choices.

Figure 2. Actor-Critic Architecture. The *critic* provides an internal reward signal to an *actor* which learns a policy for interacting with the environment.