Full (~~HTML~~) version available from the course webpage
Postscript

Journal of Artificial Intelligence Research 4 (1996) 237-285 Submitted 9/95; published 5/96

Reinforcement Learning: A Survey

Leslie Pack Kaelbling
Michael L. Littman
lpk,mlittman@cs.brown.edu
Computer Science Department, Box 1910, Brown University
Providence, RI 02912-1910 USA
and **Andrew W. Moore**
awm@cs.cmu.edu
Smith Hall 221, Carnegie Mellon University, 5000 Forbes Avenue
Pittsburgh, PA 15213 USA

Abstract

This paper surveys the field of reinforcement learning from a computer-science perspective. It is written to be accessible to researchers familiar with machine learning. Both the historical basis of the field and a broad selection of current work are summarized. Reinforcement learning is the problem faced by an agent that learns behavior through trial-and-error interactions with a dynamic environment. The work described here has a resemblance to work in psychology, but differs considerably in the details and in the use of the word "reinforcement." The paper discusses central issues of reinforcement learning, including trading off exploration and exploitation, establishing the foundations of the field via Markov decision theory, learning from delayed reinforcement, constructing empirical models to accelerate learning, making use of generalization and hierarchy, and coping with hidden state. It concludes with a survey of some implemented systems and an assessment of the practical utility of current methods for reinforcement learning.

1. Introduction

Reinforcement learning dates back to the early days of cybernetics and work in statistics, psychology, neuroscience, and computer science. In the last five to ten years, it has attracted rapidly increasing interest in the machine learning and artificial intelligence communities. Its promise is beguiling—a way of programming agents by reward and punishment without needing to specify *how* the task is to be achieved. But there are formidable computational obstacles to fulfilling the promise.

This paper surveys the historical basis of reinforcement learning and some of the current work from a computer science perspective. We give a high-level overview of the field and a taste of some specific approaches. It is, of course, impossible to mention all of the important work in the field; this should not be taken to be an exhaustive account.

Reinforcement learning is the problem faced by an agent that must learn behavior through trial-and-error interactions with a dynamic environment. The work described here has a strong family resemblance to eponymous work in psychology, but differs considerably in the details and in the use of the word "reinforcement." It is appropriately thought of as a class of problems, rather than as a set of techniques.

There are two main strategies for solving reinforcement-learning problems. The first is to search in the space of behaviors in order to find one that performs well in the environment. This approach has been taken by work in genetic algorithms and genetic programming, as well as some more novel search techniques [101]. The second is to use statistical techniques and dynamic programming methods to estimate the utility of taking actions in states of the world. This paper is devoted almost entirely to the second set of techniques because they take advantage of the special structure of reinforcement-learning problems that is not available in optimization problems in general. It is not yet clear which set of approaches is best in which circumstances.

CH4 C25

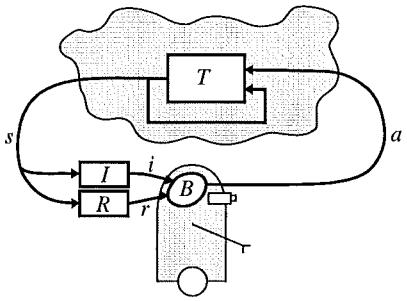

Figure 1: The standard reinforcement-learning model.

The rest of this section is devoted to establishing notation and describing the basic reinforcement-learning model. Section 2 explains the trade-off between exploration and exploitation and presents some solutions to the most basic case of reinforcement-learning problems, in which we want to maximize the immediate reward. Section 3 considers the more general problem in which rewards can be delayed in time from the actions that were crucial to gaining them. Section 4 considers some classic model-free algorithms for reinforcement learning from delayed reward: adaptive heuristic critic, $TD(\lambda)$ and Q-learning. Section 5 demonstrates a continuum of algorithms that are sensitive to the amount of computation an agent can perform between actual steps of action in the environment. Generalization—the cornerstone of mainstream machine learning research—has the potential of considerably aiding reinforcement learning, as described in Section 6. Section 7 considers the problems that arise when the agent does not have complete perceptual access to the state of the environment. Section 8 catalogs some of reinforcement learning's successful applications. Finally, Section 9 concludes with some speculations about important open problems and the future of reinforcement learning.

1.1 Reinforcement-Learning Model

In the standard reinforcement-learning model, an agent is connected to its environment via perception and action, as depicted in Figure 1. On each step of interaction the agent receives as input, i, some indication of the current state, s, of the environment; the agent then chooses an action, a, to generate as output. The action changes the state of the environment, and the value of this state transition is communicated to the agent through a scalar *reinforcement signal*, r. The agent's behavior, B, should choose actions that tend to increase the long-run sum of values of the reinforcement signal. It can learn to do this over time by systematic trial and error, guided by a wide variety of algorithms that are the subject of later sections of this paper.

Formally, the model consists of

- a discrete set of environment states, $\mathcal{S}$;
- a discrete set of agent actions, $\mathcal{A}$; and
- a set of scalar reinforcement signals; typically $\{0, 1\}$, or the real numbers.

The figure also includes an input function I, which determines how the agent views the environment state; we will assume that it is the identity function (that is, the agent perceives the exact state of the environment) until we consider partial observability in Section 7.

An intuitive way to understand the relation between the agent and its environment is with the following example dialogue.

Environment:	You are in state 65. You have 4 possible actions.
Agent:	I'll take action 2.
Environment:	You received a reinforcement of 7 units. You are now in state 15. You have 2 possible actions.
Agent:	I'll take action 1.
Environment:	You received a reinforcement of -4 units. You are now in state 65. You have 4 possible actions.
Agent:	I'll take action 2.
Environment:	You received a reinforcement of 5 units. You are now in state 44. You have 5 possible actions.
⋮	⋮

The agent's job is to find a policy π, mapping states to actions, that maximizes some long-run measure of reinforcement. We expect, in general, that the environment will be non-deterministic; that is, that taking the same action in the same state on two different occasions may result in different next states and/or different reinforcement values. This happens in our example above: from state 65, applying action 2 produces differing reinforcements and differing states on two occasions. However, we assume the environment is stationary; that is, that the *probabilities* of making state transitions or receiving specific reinforcement signals do not change over time.[1]

Reinforcement learning differs from the more widely studied problem of supervised learning in several ways. The most important difference is that there is no presentation of input/output pairs. Instead, after choosing an action the agent is told the immediate reward and the subsequent state, but is *not* told which action would have been in its best long-term interests. It is necessary for the agent to gather useful experience about the possible system states, actions, transitions and rewards actively to act optimally. Another difference from supervised learning is that on-line performance is important: the evaluation of the system is often concurrent with learning.

Some aspects of reinforcement learning are closely related to search and planning issues in artificial intelligence. AI search algorithms generate a satisfactory trajectory through a graph of states. Planning operates in a similar manner, but typically within a construct with more complexity than a graph, in which states are represented by compositions of logical expressions instead of atomic symbols. These AI algorithms are less general than the reinforcement-learning methods, in that they require a predefined model of state transitions, and with a few exceptions assume determinism. On the other hand, reinforcement learning, at least in the kind of discrete cases for which theory has been developed, assumes that the entire state space can be enumerated and stored in memory—an assumption to which conventional search algorithms are not tied.

1.2 Models of Optimal Behavior

Before we can start thinking about algorithms for learning to behave optimally, we have to decide what our model of optimality will be. In particular, we have to specify how the agent should take the future into account in the decisions it makes about how to behave now. There are three models that have been the subject of the majority of work in this area.

The *finite-horizon* model is the easiest to think about; at a given moment in time, the agent should optimize its expected reward for the next h steps:

$$E(\sum_{t=0}^{h} r_t) \ ;$$

1. This assumption may be disappointing; after all, operation in non-stationary environments is one of the motivations for building learning systems. In fact, many of the algorithms described in later sections are effective in slowly-varying non-stationary environments, but there is very little theoretical analysis in this area.

it need not worry about what will happen after that. In this and subsequent expressions, r_t represents the scalar reward received t steps into the future. This model can be used in two ways. In the first, the agent will have a non-stationary policy; that is, one that changes over time. On its first step it will take what is termed a *h-step optimal action.* This is defined to be the best action available given that it has h steps remaining in which to act and gain reinforcement. On the next step it will take a $(h-1)$-step optimal action, and so on, until it finally takes a 1-step optimal action and terminates. In the second, the agent does *receding-horizon control*, in which it always takes the h-step optimal action. The agent always acts according to the same policy, but the value of h limits how far ahead it looks in choosing its actions. The finite-horizon model is not always appropriate. In many cases we may not know the precise length of the agent's life in advance.

The infinite-horizon discounted model takes the long-run reward of the agent into account, but rewards that are received in the future are geometrically discounted according to discount factor γ, (where $0 \leq \gamma < 1$):

$$E(\sum_{t=0}^{\infty} \gamma^t r_t) \ .$$

We can interpret γ in several ways. It can be seen as an interest rate, a probability of living another step, or as a mathematical trick to bound the infinite sum. The model is conceptually similar to receding-horizon control, but the discounted model is more mathematically tractable than the finite-horizon model. This is a dominant reason for the wide attention this model has received.

Another optimality criterion is the *average-reward model*, in which the agent is supposed to take actions that optimize its long-run average reward:

$$\lim_{h \to \infty} E(\frac{1}{h} \sum_{t=0}^{h} r_t) \ .$$

Such a policy is referred to as a *gain optimal* policy; it can be seen as the limiting case of the infinite-horizon discounted model as the discount factor approaches 1 [14]. One problem with this criterion is that there is no way to distinguish between two policies, one of which gains a large amount of reward in the initial phases and the other of which does not. Reward gained on any initial prefix of the agent's life is overshadowed by the long-run average performance. It is possible to generalize this model so that it takes into account both the long run average and the amount of initial reward than can be gained. In the generalized, *bias optimal* model, a policy is preferred if it maximizes the long-run average and ties are broken by the initial extra reward.

Figure 2 contrasts these models of optimality by providing an environment in which changing the model of optimality changes the optimal policy. In this example, circles represent the states of the environment and arrows are state transitions. There is only a single action choice from every state except the start state, which is in the upper left and marked with an incoming arrow. All rewards are zero except where marked. Under a finite-horizon model with $h = 5$, the three actions yield rewards of +6.0, +0.0, and +0.0, so the first action should be chosen; under an infinite-horizon discounted model with $\gamma = 0.9$, the three choices yield +16.2, +59.0, and +58.5 so the second action should be chosen; and under the average reward model, the third action should be chosen since it leads to an average reward of +11. If we change h to 1000 and γ to 0.2, then the second action is optimal for the finite-horizon model and the first for the infinite-horizon discounted model; however, the average reward model will always prefer the best long-term average. Since the choice of optimality model and parameters matters so much, it is important to choose it carefully in any application.

The finite-horizon model is appropriate when the agent's lifetime is known; one important aspect of this model is that as the length of the remaining lifetime decreases, the agent's policy may change. A system with a hard deadline would be appropriately modeled this way. The relative usefulness of infinite-horizon discounted and bias-optimal models is still under debate. Bias-optimality has the advantage of not requiring a discount parameter; however, algorithms for finding bias-optimal policies are not yet as well-understood as those for finding optimal infinite-horizon discounted policies.

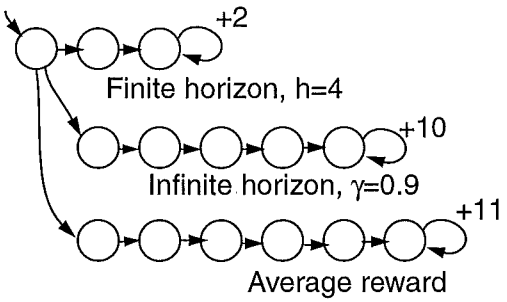

Figure 2: Comparing models of optimality. All unlabeled arrows produce a reward of zero.

1.3 Measuring Learning Performance

The criteria given in the previous section can be used to assess the policies learned by a given algorithm. We would also like to be able to evaluate the quality of learning itself. There are several incompatible measures in use.

- **Eventual convergence to optimal.** Many algorithms come with a provable guarantee of asymptotic convergence to optimal behavior [129]. This is reassuring, but useless in practical terms. An agent that quickly reaches a plateau at 99% of optimality may, in many applications, be preferable to an agent that has a guarantee of eventual optimality but a sluggish early learning rate.

- **Speed of convergence to optimality.** Optimality is usually an asymptotic result, and so convergence speed is an ill-defined measure. More practical is the *speed of convergence to near-optimality.* This measure begs the definition of how near to optimality is sufficient. A related measure is *level of performance after a given time*, which similarly requires that someone define the given time.

  It should be noted that here we have another difference between reinforcement learning and conventional supervised learning. In the latter, expected future predictive accuracy or statistical efficiency are the prime concerns. For example, in the well-known PAC framework [127], there is a learning period during which mistakes do not count, then a performance period during which they do. The framework provides bounds on the necessary length of the learning period in order to have a probabilistic guarantee on the subsequent performance. That is usually an inappropriate view for an agent with a long existence in a complex environment.

  In spite of the mismatch between embedded reinforcement learning and the train/test perspective, Fiechter [39] provides a PAC analysis for Q-learning (described in Section 4.2) that sheds some light on the connection between the two views.

  Measures related to speed of learning have an additional weakness. An algorithm that merely tries to achieve optimality as fast as possible may incur unnecessarily large penalties during the learning period. A less aggressive strategy taking longer to achieve optimality, but gaining greater total reinforcement during its learning might be preferable.

- **Regret.** A more appropriate measure, then, is the expected decrease in reward gained due to executing the learning algorithm instead of behaving optimally from the very beginning. This measure is known as *regret* [12]. It penalizes mistakes wherever they occur during the run. Unfortunately, results concerning the regret of algorithms are quite hard to obtain.

546 <29

1.4 Reinforcement Learning and Adaptive Control

Adaptive control [19, 112] is also concerned with algorithms for improving a sequence of decisions from experience. Adaptive control is a much more mature discipline that concerns itself with dynamic systems in which states and actions are vectors and system dynamics are smooth: linear or locally linearizable around a desired trajectory. A very common formulation of cost functions in adaptive control are quadratic penalties on deviation from desired state and action vectors. Most importantly, although the dynamic model of the system is not known in advance, and must be estimated from data, the *structure* of the dynamic model is fixed, leaving model estimation as a parameter estimation problem. These assumptions permit deep, elegant and powerful mathematical analysis, which in turn lead to robust, practical, and widely deployed adaptive control algorithms.

2. Exploitation versus Exploration: The Single-State Case

One major difference between reinforcement learning and supervised learning is that a reinforcement-learner must explicitly explore its environment. In order to highlight the problems of exploration, we treat a very simple case in this section. The fundamental issues and approaches described here will, in many cases, transfer to the more complex instances of reinforcement learning discussed later in the paper.

The simplest possible reinforcement-learning problem is known as the k-armed bandit problem, which has been the subject of a great deal of study in the statistics and applied mathematics literature [12]. The agent is in a room with a collection of k gambling machines (each called a "one-armed bandit" in colloquial English). The agent is permitted a fixed number of pulls, h. Any arm may be pulled on each turn. The machines do not require a deposit to play; the only cost is in wasting a pull playing a suboptimal machine. When arm i is pulled, machine i pays off 1 or 0, according to some underlying probability parameter p_i, where payoffs are independent events and the p_is are unknown. What should the agent's strategy be?

This problem illustrates the fundamental tradeoff between exploitation and exploration. The agent might believe that a particular arm has a fairly high payoff probability; should it choose that arm all the time, or should it choose another one that it has less information about, but seems to be worse? Answers to these questions depend on how long the agent is expected to play the game; the longer the game lasts, the worse the consequences of prematurely converging on a sub-optimal arm, and the more the agent should explore.

There is a wide variety of solutions to this problem. We will consider a representative selection of them, but for a deeper discussion and a number of important theoretical results, see the book by Berry and Fristedt [12]. We use the term "action" to indicate the agent's choice of arm to pull. This eases the transition into delayed reinforcement models in Section 3. It is very important to note that bandit problems fit our definition of a reinforcement-learning environment with a single state with only self transitions.

Section 2.1 discusses three solutions to the basic one-state bandit problem that have formal correctness results. Although they can be extended to problems with real-valued rewards, they do not apply directly to the general multi-state delayed-reinforcement case. Section 2.2 presents three techniques that are not formally justified, but that have had wide use in practice, and can be applied (with similar lack of guarantee) to the general case.

2.1 Formally Justified Techniques

There is a fairly well-developed formal theory of exploration for very simple problems. Although it is instructive, the methods it provides do not scale well to more complex problems.

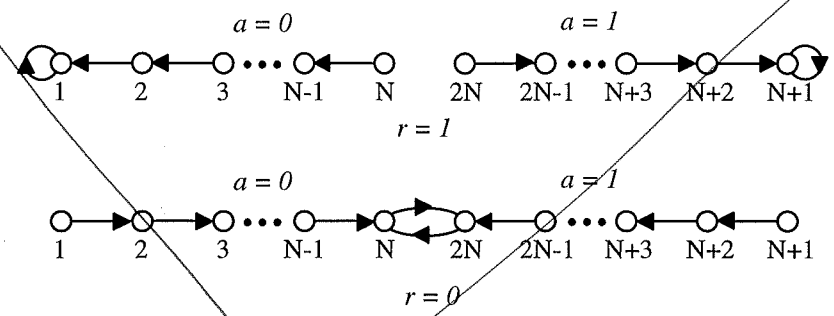

Figure 3: A Tsetlin automaton with $2N$ states. The top row shows the state transitions that are made when the previous action resulted in a reward of 1; the bottom row shows transitions after a reward of 0. In states in the left half of the figure, action 0 is taken; in those on the right, action 1 is taken.

- When action a_i succeeds,

$$\begin{aligned} p_i &:= p_i + \alpha(1 - p_i) \\ p_j &:= p_j - \alpha p_j \text{ for } j \neq i \end{aligned}$$

- When action a_i fails, p_j remains unchanged (for all j).

This algorithm converges with probability 1 to a vector containing a single 1 and the rest 0's (choosing a particular action with probability 1). Unfortunately, it does not always converge to the correct action; but the probability that it converges to the wrong one can be made arbitrarily small by making α small [86]. There is no literature on the regret of this algorithm.

2.2 Ad-Hoc Techniques

In reinforcement-learning practice, some simple, *ad hoc* strategies have been popular. They are rarely, if ever, the best choice for the models of optimality we have used, but they may be viewed as reasonable, computationally tractable, heuristics. Thrun [124] has surveyed a variety of these techniques.

2.2.1 GREEDY STRATEGIES

The first strategy that comes to mind is to always choose the action with the highest estimated payoff. The flaw is that early unlucky sampling might indicate that the best action's reward is less than the reward obtained from a suboptimal action. The suboptimal action will always be picked, leaving the true optimal action starved of data and its superiority never discovered. An agent must explore to ameliorate this outcome.

A useful heuristic is *optimism in the face of uncertainty* in which actions are selected greedily, but strongly optimistic prior beliefs are put on their payoffs so that strong negative evidence is needed to eliminate an action from consideration. This still has a measurable danger of starving an optimal but unlucky action, but the risk of this can be made arbitrarily small. Techniques like this have been used in several reinforcement learning algorithms including the interval exploration method [52] (described shortly), the *exploration bonus* in Dyna [116], *curiosity-driven exploration* [102], and the exploration mechanism in prioritized sweeping [83].

C48 C31

2.2.2 RANDOMIZED STRATEGIES

Another simple exploration strategy is to take the action with the best estimated expected reward by default, but with probability p, choose an action at random. Some versions of this strategy start with a large value of p to encourage initial exploration, which is slowly decreased.

An objection to the simple strategy is that when it experiments with a non-greedy action it is no more likely to try a promising alternative than a clearly hopeless alternative. A slightly more sophisticated strategy is *Boltzmann exploration.* In this case, the expected reward for taking action a, $ER(a)$ is used to choose an action probabilistically according to the distribution

$$P(a) = \frac{e^{ER(a)/T}}{\sum_{a' \in A} e^{ER(a')/T}} \cdot$$

The *temperature* parameter T can be decreased over time to decrease exploration. This method works well if the best action is well separated from the others, but suffers somewhat when the values of the actions are close. It may also converge unnecessarily slowly unless the temperature schedule is manually tuned with great care.

2.2.3 INTERVAL-BASED TECHNIQUES

Exploration is often more efficient when it is based on second-order information about the certainty or variance of the estimated values of actions. Kaelbling's *interval estimation* algorithm [52] stores statistics for each action a_i: w_i is the number of successes and n_i the number of trials. An action is chosen by computing the upper bound of a $100 \cdot (1 - \alpha)\%$ confidence interval on the success probability of each action and choosing the action with the highest upper bound. Smaller values of the α parameter encourage greater exploration. When payoffs are boolean, the normal approximation to the binomial distribution can be used to construct the confidence interval (though the binomial should be used for small n). Other payoff distributions can be handled using their associated statistics or with nonparametric methods. The method works very well in empirical trials. It is also related to a certain class of statistical techniques known as *experiment design* methods [17], which are used for comparing multiple treatments (for example, fertilizers or drugs) to determine which treatment (if any) is best in as small a set of experiments as possible.

2.3 More General Problems

When there are multiple states, but reinforcement is still immediate, then any of the above solutions can be replicated, once for each state. However, when generalization is required, these solutions must be integrated with generalization methods (see section 6); this is straightforward for the simple ad-hoc methods, but it is not understood how to maintain theoretical guarantees.

Many of these techniques focus on converging to some regime in which exploratory actions are taken rarely or never; this is appropriate when the environment is stationary. However, when the environment is non-stationary, exploration must continue to take place, in order to notice changes in the world. Again, the more ad-hoc techniques can be modified to deal with this in a plausible manner (keep temperature parameters from going to 0; decay the statistics in interval estimation), but none of the theoretically guaranteed methods can be applied.

3. Delayed Reward

In the general case of the reinforcement learning problem, the agent's actions determine not only its immediate reward, but also (at least probabilistically) the next state of the environment. Such environments can be thought of as networks of bandit problems, but the agent must take into account the next state as well as the immediate reward when it decides which action to take. The model of long-run optimality the agent is using determines exactly how it should take the value of the future

C49 C32

into account. The agent will have to be able to learn from delayed reinforcement: it may take a long sequence of actions, receiving insignificant reinforcement, then finally arrive at a state with high reinforcement. The agent must be able to learn which of its actions are desirable based on reward that can take place arbitrarily far in the future.

3.1 Markov Decision Processes

Problems with delayed reinforcement are well modeled as *Markov decision processes* (MDPs). An MDP consists of

- a set of states $\mathcal{S}$,
- a set of actions $\mathcal{A}$,
- a reward function $R : \mathcal{S} \times \mathcal{A} \rightarrow \Re$, and
- a state transition function $T : \mathcal{S} \times \mathcal{A} \rightarrow \Pi(\mathcal{S})$, where a member of $\Pi(\mathcal{S})$ is a probability distribution over the set $\mathcal{S}$ (i.e. it maps states to probabilities). We write $T(s, a, s')$ for the probability of making a transition from state s to state s' using action a.

The state transition function probabilistically specifies the next state of the environment as a function of its current state and the agent's action. The reward function specifies expected instantaneous reward as a function of the current state and action. The model is *Markov* if the state transitions are independent of any previous environment states or agent actions. There are many good references to MDP models [10, 13, 48, 90].

Although general MDPs may have infinite (even uncountable) state and action spaces, we will only discuss methods for solving finite-state and finite-action problems. In section 6, we discuss methods for solving problems with continuous input and output spaces.

3.2 Finding a Policy Given a Model

Before we consider algorithms for learning to behave in MDP environments, we will explore techniques for determining the optimal policy given a correct model. These dynamic programming techniques will serve as the foundation and inspiration for the learning algorithms to follow. We restrict our attention mainly to finding optimal policies for the infinite-horizon discounted model, but most of these algorithms have analogs for the finite-horizon and average-case models as well. We rely on the result that, for the infinite-horizon discounted model, there exists an optimal deterministic stationary policy [10].

We will speak of the optimal *value* of a state—it is the expected infinite discounted sum of reward that the agent will gain if it starts in that state and executes the optimal policy. Using π as a complete decision policy, it is written

$$V^*(s) = \max_\pi E\left(\sum_{t=0}^{\infty} \gamma^t r_t\right) \quad .$$

This optimal value function is unique and can be defined as the solution to the simultaneous equations

$$V^*(s) = \max_a \left(R(s,a) + \gamma \sum_{s' \in \mathcal{S}} T(s,a,s')V^*(s')\right), \forall s \in \mathcal{S} \quad , \tag{1}$$

which assert that the value of a state s is the expected instantaneous reward plus the expected discounted value of the next state, using the best available action. Given the optimal value function,

C50 C33

we can specify the optimal policy as

$$\pi^*(s) = \arg\max_a \left(R(s,a) + \gamma \sum_{s' \in \mathcal{S}} T(s,a,s') V^*(s') \right) .$$

3.2.1 VALUE ITERATION

One way, then, to find an optimal policy is to find the optimal value function. It can be determined by a simple iterative algorithm called *value iteration* that can be shown to converge to the correct V^* values [10, 13].

```
initialize V(s) arbitrarily
loop until policy good enough
    loop for s ∈ S
        loop for a ∈ A
            Q(s,a) := R(s,a) + γ Σ_{s'∈S} T(s,a,s')V(s')
        V(s) := max_a Q(s,a)
    end loop
end loop
```

It is not obvious when to stop the value iteration algorithm. One important result bounds the performance of the current greedy policy as a function of the *Bellman residual* of the current value function [134]. It says that if the maximum difference between two successive value functions is less than ϵ, then the value of the greedy policy, (the policy obtained by choosing, in every state, the action that maximizes the estimated discounted reward, using the current estimate of the value function) differs from the value function of the optimal policy by no more than $2\epsilon\gamma/(1-\gamma)$ at any state. This provides an effective stopping criterion for the algorithm. Puterman [90] discusses another stopping criterion, based on the *span semi-norm*, which may result in earlier termination. Another important result is that the greedy policy is guaranteed to be optimal in some finite number of steps even though the value function may not have converged [13]. And in practice, the greedy policy is often optimal long before the value function has converged.

Value iteration is very flexible. The assignments to V need not be done in strict order as shown above, but instead can occur asynchronously in parallel provided that the value of every state gets updated infinitely often on an infinite run. These issues are treated extensively by Bertsekas [16], who also proves convergence results.

Updates based on Equation 1 are known as *full backups* since they make use of information from all possible successor states. It can be shown that updates of the form

$$Q(s,a) := Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$$

can also be used as long as each pairing of a and s is updated infinitely often, s' is sampled from the distribution $T(s,a,s')$, r is sampled with mean $R(s,a)$ and bounded variance, and the learning rate α is decreased slowly. This type of *sample backup* [111] is critical to the operation of the model-free methods discussed in the next section.

The computational complexity of the value-iteration algorithm with full backups, per iteration, is quadratic in the number of states and linear in the number of actions. Commonly, the transition probabilities $T(s,a,s')$ are sparse. If there are on average a constant number of next states with non-zero probability then the cost per iteration is linear in the number of states and linear in the number of actions. The number of iterations required to reach the optimal value function is polynomial in the number of states and the magnitude of the largest reward if the discount factor is held constant. However, in the worst case the number of iterations grows polynomially in $1/(1-\gamma)$, so the convergence rate slows considerably as the discount factor approaches 1 [66].

C54 C34

3.2.2 POLICY ITERATION

The *policy iteration* algorithm manipulates the policy directly, rather than finding it indirectly via the optimal value function. It operates as follows:

```
choose an arbitrary policy π'
loop
    π := π'
    compute the value function of policy π:
        solve the linear equations
            V_π(s) = R(s, π(s)) + γ Σ_{s'∈S} T(s, π(s), s') V_π(s')
    improve the policy at each state:
        π'(s) := arg max_a (R(s, a) + γ Σ_{s'∈S} T(s, a, s') V_π(s'))
until π = π'
```

The value function of a policy is just the expected infinite discounted reward that will be gained, at each state, by executing that policy. It can be determined by solving a set of linear equations. Once we know the value of each state under the current policy, we consider whether the value could be improved by changing the first action taken. If it can, we change the policy to take the new action whenever it is in that situation. This step is guaranteed to strictly improve the performance of the policy. When no improvements are possible, then the policy is guaranteed to be optimal.

Since there are at most $|\mathcal{A}|^{|\mathcal{S}|}$ distinct policies, and the sequence of policies improves at each step, this algorithm terminates in at most an exponential number of iterations [90]. However, it is an important open question how many iterations policy iteration takes in the worst case. It is known that the running time is pseudopolynomial and that for any fixed discount factor, there is a polynomial bound in the total size of the MDP [66].

3.2.3 ENHANCEMENT TO VALUE ITERATION AND POLICY ITERATION

In practice, value iteration is much faster per iteration, but policy iteration takes fewer iterations. Arguments have been put forth to the effect that each approach is better for large problems. Puterman's *modified policy iteration* algorithm [91] provides a method for trading iteration time for iteration improvement in a smoother way. The basic idea is that the expensive part of policy iteration is solving for the exact value of V_π. Instead of finding an exact value for V_π, we can perform a few steps of a modified value-iteration step where the policy is held fixed over successive iterations. This can be shown to produce an approximation to V_π that converges linearly in γ. In practice, this can result in substantial speedups.

Several standard numerical-analysis techniques that speed the convergence of dynamic programming can be used to accelerate value and policy iteration. *Multigrid methods* can be used to quickly seed a good initial approximation to a high resolution value function by initially performing value iteration at a coarser resolution [93]. *State aggregation* works by collapsing groups of states to a single meta-state solving the abstracted problem [15].

3.2.4 COMPUTATIONAL COMPLEXITY

Value iteration works by producing successive approximations of the optimal value function. Each iteration can be performed in $O(|A||S|^2)$ steps, or faster if there is sparsity in the transition function. However, the number of iterations required can grow exponentially in the discount factor [27]; as the discount factor approaches 1, the decisions must be based on results that happen farther and farther into the future. In practice, policy iteration converges in fewer iterations than value iteration, although the per-iteration costs of $O(|A||S|^2 + |S|^3)$ can be prohibitive. There is no known tight worst-case bound available for policy iteration [66]. Modified policy iteration [91] seeks a trade-off between cheap and effective iterations and is preferred by some practictioners [96].

C57 C35

Linear programming [105] is an extremely general problem, and MDPs can be solved by general-purpose linear-programming packages [35, 34, 46]. An advantage of this approach is that commercial-quality linear-programming packages are available, although the time and space requirements can still be quite high. From a theoretic perspective, linear programming is the only known algorithm that can solve MDPs in polynomial time, although the theoretically efficient algorithms have not been shown to be efficient in practice.

4. Learning an Optimal Policy: Model-free Methods

In the previous section we reviewed methods for obtaining an optimal policy for an MDP assuming that we already had a model. The model consists of knowledge of the state transition probability function $T(s, a, s')$ and the reinforcement function $R(s, a)$. Reinforcement learning is primarily concerned with how to obtain the optimal policy when such a model is not known in advance. The agent must interact with its environment directly to obtain information which, by means of an appropriate algorithm, can be processed to produce an optimal policy.

At this point, there are two ways to proceed.

- **Model-free:** Learn a controller without learning a model.
- **Model-based:** Learn a model, and use it to derive a controller.

Which approach is better? This is a matter of some debate in the reinforcement-learning community. A number of algorithms have been proposed on both sides. This question also appears in other fields, such as adaptive control, where the dichotomy is between *direct* and *indirect* adaptive control.

This section examines model-free learning, and Section 5 examines model-based methods.

The biggest problem facing a reinforcement-learning agent is *temporal credit assignment*. How do we know whether the action just taken is a good one, when it might have far-reaching effects? One strategy is to wait until the "end" and reward the actions taken if the result was good and punish them if the result was bad. In ongoing tasks, it is difficult to know what the "end" is, and this might require a great deal of memory. Instead, we will use insights from value iteration to adjust the estimated value of a state based on the immediate reward and the estimated value of the next state. This class of algorithms is known as *temporal difference methods* [115]. We will consider two different temporal-difference learning strategies for the discounted infinite-horizon model.

4.1 Adaptive Heuristic Critic and TD(λ)

The *adaptive heuristic critic* algorithm is an adaptive version of policy iteration [9] in which the value-function computation is no longer implemented by solving a set of linear equations, but is instead computed by an algorithm called $TD(0)$. A block diagram for this approach is given in Figure 4. It consists of two components: a critic (labeled AHC), and a reinforcement-learning component (labeled RL). The reinforcement-learning component can be an instance of any of the k-armed bandit algorithms, modified to deal with multiple states and non-stationary rewards. But instead of acting to maximize instantaneous reward, it will be acting to maximize the heuristic value, v, that is computed by the critic. The critic uses the real external reinforcement signal to learn to map states to their expected discounted values given that the policy being executed is the one currently instantiated in the RL component.

We can see the analogy with modified policy iteration if we imagine these components working in alternation. The policy π implemented by RL is fixed and the critic learns the value function V_π for that policy. Now we fix the critic and let the RL component learn a new policy π' that maximizes the new value function, and so on. In most implementations, however, both components operate simultaneously. Only the alternating implementation can be guaranteed to converge to the optimal policy, under appropriate conditions. Williams and Baird explored the convergence properties of a class of AHC-related algorithms they call "incremental variants of policy iteration" [133].

C53 C36

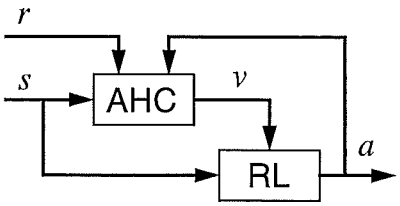

Figure 4: Architecture for the adaptive heuristic critic.

It remains to explain how the critic can learn the value of a policy. We define $\langle s, a, r, s' \rangle$ to be an *experience tuple* summarizing a single transition in the environment. Here s is the agent's state before the transition, a is its choice of action, r the instantaneous reward it receives, and s' its resulting state. The value of a policy is learned using Sutton's $TD(0)$ algorithm [115] which uses the update rule

$$V(s) := V(s) + \alpha(r + \gamma V(s') - V(s)) \ .$$

Whenever a state s is visited, its estimated value is updated to be closer to $r + \gamma V(s')$, since r is the instantaneous reward received and $V(s')$ is the estimated value of the actually occurring next state. This is analogous to the sample-backup rule from value iteration—the only difference is that the sample is drawn from the real world rather than by simulating a known model. The key idea is that $r + \gamma V(s')$ is a sample of the value of $V(s)$, and it is more likely to be correct because it incorporates the real r. If the learning rate α is adjusted properly (it must be slowly decreased) and the policy is held fixed, $TD(0)$ is guaranteed to converge to the optimal value function.

The $TD(0)$ rule as presented above is really an instance of a more general class of algorithms called $TD(\lambda)$, with $\lambda = 0$. $TD(0)$ looks only one step ahead when adjusting value estimates; although it will eventually arrive at the correct answer, it can take quite a while to do so. The general $TD(\lambda)$ rule is similar to the $TD(0)$ rule given above,

$$V(u) := V(u) + \alpha(r + \gamma V(s') - V(s))e(u) \ ,$$

but it is applied to *every state* according to its eligibility $e(u)$, rather than just to the immediately previous state, s. One version of the eligibility trace is defined to be

$$e(s) = \sum_{k=1}^{t} (\lambda\gamma)^{t-k} \delta_{s,s_k} \ , \text{ where } \delta_{s,s_k} = \begin{cases} 1 \text{ if } s = s_k \\ 0 \text{ otherwise} \end{cases} \ .$$

The eligibility of a state s is the degree to which it has been visited in the recent past; when a reinforcement is received, it is used to update all the states that have been recently visited, according to their eligibility. When $\lambda = 0$ this is equivalent to $TD(0)$. When $\lambda = 1$, it is roughly equivalent to updating all the states according to the number of times they were visited by the end of a run. Note that we can update the eligibility online as follows:

$$e(s) := \begin{cases} \gamma\lambda e(s) + 1 & \text{if } s = \text{ current state} \\ \gamma\lambda e(s) & \text{otherwise} \end{cases} \ .$$

It is computationally more expensive to execute the general $TD(\lambda)$, though it often converges considerably faster for large λ [30, 32]. There has been some recent work on making the updates more efficient [24] and on changing the definition to make $TD(\lambda)$ more consistent with the certainty-equivalent method [108], which is discussed in Section 5.1.

4.2 Q-learning

The work of the two components of AHC can be accomplished in a unified manner by Watkins' Q-learning algorithm [128, 129]. Q-learning is typically easier to implement. In order to understand

C54 C37

Q-learning, we have to develop some additional notation. Let $Q^*(s,a)$ be the expected discounted reinforcement of taking action a in state s, then continuing by choosing actions optimally. Note that $V^*(s)$ is the value of s assuming the best action is taken initially, and so $V^*(s) = \max_a Q^*(s,a)$. $Q^*(s,a)$ can hence be written recursively as

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') \max_{a'} Q^*(s',a') \ .$$

Note also that, since $V^*(s) = \max_a Q^*(s,a)$, we have $\pi^*(s) = \arg\max_a Q^*(s,a)$ as an optimal policy.

Because the Q function makes the action explicit, we can estimate the Q values on-line using a method essentially the same as $TD(0)$, but also use them to define the policy, because an action can be chosen just by taking the one with the maximum Q value for the current state.

The Q-learning rule is

$$Q(s,a) := Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a)) \ ,$$

where $\langle s,a,r,s' \rangle$ is an experience tuple as described earlier. If each action is executed in each state an infinite number of times on an infinite run and α is decayed appropriately, the Q values will converge with probability 1 to Q^* [128, 125, 49]. Q-learning can also be extended to update states that occurred more than one step previously, as in $TD(\lambda)$ [88].

When the Q values are nearly converged to their optimal values, it is appropriate for the agent to act greedily, taking, in each situation, the action with the highest Q value. During learning, however, there is a difficult exploitation versus exploration trade-off to be made. There are no good, formally justified approaches to this problem in the general case; standard practice is to adopt one of the *ad hoc* methods discussed in section 2.2.

AHC architectures seem to be more difficult to work with than Q-learning on a practical level. It can be hard to get the relative learning rates right in AHC so that the two components converge together. In addition, Q-learning is *exploration insensitive*: that is, that the Q values will converge to the optimal values, independent of how the agent behaves while the data is being collected (as long as all state-action pairs are tried often enough). This means that, although the exploration-exploitation issue must be addressed in Q-learning, the details of the exploration strategy will not affect the convergence of the learning algorithm. For these reasons, Q-learning is the most popular and seems to be the most effective model-free algorithm for learning from delayed reinforcement. It does not, however, address any of the issues involved in generalizing over large state and/or action spaces. In addition, it may converge quite slowly to a good policy.

4.3 Model-free Learning With Average Reward

As described, Q-learning can be applied to discounted infinite-horizon MDPs. It can also be applied to undiscounted problems as long as the optimal policy is guaranteed to reach a reward-free absorbing state and the state is periodically reset.

Schwartz [106] examined the problem of adapting Q-learning to an average-reward framework. Although his R-learning algorithm seems to exhibit convergence problems for some MDPs, several researchers have found the average-reward criterion closer to the true problem they wish to solve than a discounted criterion and therefore prefer R-learning to Q-learning [69].

With that in mind, researchers have studied the problem of learning optimal average-reward policies. Mahadevan [70] surveyed model-based average-reward algorithms from a reinforcement-learning perspective and found several difficulties with existing algorithms. In particular, he showed that existing reinforcement-learning algorithms for average reward (and some dynamic programming algorithms) do not always produce bias-optimal policies. Jaakkola, Jordan and Singh [50] described an average-reward learning algorithm with guaranteed convergence properties. It uses a Monte-Carlo component to estimate the expected future reward for each state as the agent moves through the environment. In addition, Bertsekas presents a Q-learning-like algorithm for average-case reward in his new textbook [14]. Although this recent work provides a much needed theoretical foundation to this area of reinforcement learning, many important problems remain unsolved.

If we have updated the V value for state s' and it has changed by amount Δ, then the immediate predecessors of s' are informed of this event. Any state s for which there exists an action a such that $\hat{T}(s,a,s') \neq 0$ has its priority promoted to $\Delta \cdot \hat{T}(s,a,s')$, unless its priority already exceeded that value.

The global behavior of this algorithm is that when a real-world transition is "surprising" (the agent happens upon a goal state, for instance), then lots of computation is directed to propagate this new information back to relevant predecessor states. When the real-world transition is "boring" (the actual result is very similar to the predicted result), then computation continues in the most deserving part of the space.

Running prioritized sweeping on the problem in Figure 6, we see a large improvement over Dyna. The optimal policy is reached in about half the number of steps of experience and one-third the computation as Dyna required (and therefore about 20 times fewer steps and twice the computational effort of Q-learning).

5.4 Other Model-Based Methods

Methods proposed for solving MDPs given a model can be used in the context of model-based methods as well.

RTDP (real-time dynamic programming) [8] is another model-based method that uses Q-learning to concentrate computational effort on the areas of the state-space that the agent is most likely to occupy. It is specific to problems in which the agent is trying to achieve a particular goal state and the reward everywhere else is 0. By taking into account the start state, it can find a short path from the start to the goal, without necessarily visiting the rest of the state space.

The Plexus planning system [33, 55] exploits a similar intuition. It starts by making an approximate version of the MDP which is much smaller than the original one. The approximate MDP contains a set of states, called the *envelope*, that includes the agent's current state and the goal state, if there is one. States that are not in the envelope are summarized by a single "out" state. The planning process is an alternation between finding an optimal policy on the approximate MDP and adding useful states to the envelope. Action may take place in parallel with planning, in which case irrelevant states are also pruned out of the envelope.

6. Generalization

All of the previous discussion has tacitly assumed that it is possible to enumerate the state and action spaces and store tables of values over them. Except in very small environments, this means impractical memory requirements. It also makes inefficient use of experience. In a large, smooth state space we generally expect similar states to have similar values and similar optimal actions. Surely, therefore, there should be some more compact representation than a table. Most problems will have continuous or large discrete state spaces; some will have large or continuous action spaces. The problem of learning in large spaces is addressed through *generalization techniques*, which allow compact storage of learned information and transfer of knowledge between "similar" states and actions.

The large literature of generalization techniques from inductive concept learning can be applied to reinforcement learning. However, techniques often need to be tailored to specific details of the problem. In the following sections, we explore the application of standard function-approximation techniques, adaptive resolution models, and hierarchical methods to the problem of reinforcement learning.

The reinforcement-learning architectures and algorithms discussed above have included the storage of a variety of mappings, including $\mathcal{S} \rightarrow \mathcal{A}$ (policies), $\mathcal{S} \rightarrow \Re$ (value functions), $\mathcal{S} \times \mathcal{A} \rightarrow \Re$ (Q functions and rewards), $\mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ (deterministic transitions), and $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0,1]$ (transition probabilities). Some of these mappings, such as transitions and immediate rewards, can be learned using straightforward supervised learning, and can be handled using any of the wide variety of function-approximation techniques for supervised learning that support noisy training examples. Popular techniques include various neural-network methods [94], fuzzy logic [11, 58], CMAC [3], and local memory-based methods [84], such as generalizations of nearest neighbor methods. Other mappings, especially the policy mapping, typically need specialized algorithms because training sets of input-output pairs are not available.

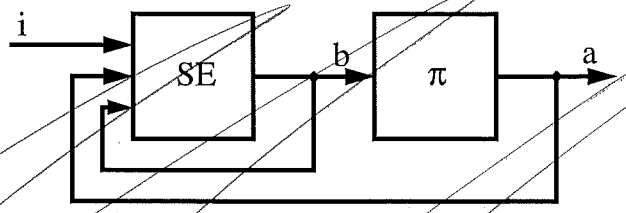

Figure 10: Structure of a POMDP agent.

belief state, the last action a, and the current observation i. In this context, a belief state is a probability distribution over states of the environment, indicating the likelihood, given the agent's past experience, that the environment is actually in each of those states. The state estimator can be constructed straightforwardly using the estimated world model and Bayes' rule.

Now we are left with the problem of finding a policy mapping belief states into action. This problem can be formulated as an MDP, but it is difficult to solve using the techniques described earlier, because the input space is continuous. Chrisman's approach [22] does not take into account future uncertainty, but yields a policy after a small amount of computation. A standard approach from the operations-research literature is to solve for the optimal policy (or a close approximation thereof) based on its representation as a piecewise-linear and convex function over the belief space. This method is computationally intractable, but may serve as inspiration for methods that make further approximations [20, 65].

8. Reinforcement Learning Applications

One reason that reinforcement learning is popular is that is serves as a theoretical tool for studying the principles of agents learning to act. But it is unsurprising that it has also been used by a number of researchers as a practical computational tool for constructing autonomous systems that improve themselves with experience. These applications have ranged from robotics, to industrial manufacturing, to combinatorial search problems such as computer game playing.

Practical applications provide a test of the efficacy and usefulness of learning algorithms. They are also an inspiration for deciding which components of the reinforcement learning framework are of practical importance. For example, a researcher with a real robotic task can provide a data point to questions such as:

- How important is optimal exploration? Can we break the learning period into exploration phases and exploitation phases?
- What is the most useful model of long-term reward: Finite horizon? Discounted? Infinite horizon?
- How much computation is available between agent decisions and how should it be used?
- What prior knowledge can we build into the system, and which algorithms are capable of using that knowledge?

Let us examine a set of practical applications of reinforcement learning, while bearing these questions in mind.

8.1 Game Playing

Game playing has dominated the Artificial Intelligence world as a problem domain ever since the field was born. Two-player games do not fit into the established reinforcement-learning framework

CS7 C40

since the optimality criterion for games is not one of maximizing reward in the face of a fixed environment, but one of maximizing reward against an optimal adversary (minimax). Nonetheless, reinforcement-learning algorithms can be adapted to work for a very general class of games [63] and many researchers have used reinforcement learning in these environments. One application, spectacularly far ahead of its time, was Samuel's checkers playing system [99]. This learned a value function represented by a linear function approximator, and employed a training scheme similar to the updates used in value iteration, temporal differences and Q-learning.

More recently, Tesauro [118, 119, 120] applied the temporal difference algorithm to backgammon. Backgammon has approximately 10^{20} states, making table-based reinforcement learning impossible. Instead, Tesauro used a backpropagation-based three-layer neural network as a function approximator for the value function

Board Position $\rightarrow$ *Probability of victory for current player.*

Two versions of the learning algorithm were used. The first, which we will call Basic TD-Gammon, used very little predefined knowledge of the game, and the representation of a board position was virtually a raw encoding, sufficiently powerful only to permit the neural network to distinguish between conceptually different positions. The second, TD-Gammon, was provided with the same raw state information supplemented by a number of hand-crafted features of backgammon board positions. Providing hand-crafted features in this manner is a good example of how inductive biases from human knowledge of the task can be supplied to a learning algorithm.

The training of both learning algorithms required several months of computer time, and was achieved by constant self-play. No exploration strategy was used—the system always greedily chose the move with the largest expected probability of victory. This naive exploration strategy proved entirely adequate for this environment, which is perhaps surprising given the considerable work in the reinforcement-learning literature which has produced numerous counter-examples to show that greedy exploration can lead to poor learning performance. Backgammon, however, has two important properties. Firstly, whatever policy is followed, every game is guaranteed to end in finite time, meaning that useful reward information is obtained fairly frequently. Secondly, the state transitions are sufficiently stochastic that independent of the policy, all states will occasionally be visited—a wrong initial value function has little danger of starving us from visiting a critical part of state space from which important information could be obtained.

The results (Table 2) of TD-Gammon are impressive. It has competed at the very top level of international human play. Basic TD-Gammon played respectably, but not at a professional standard.

Although experiments with other games have in some cases produced interesting learning behavior, no success close to that of TD-Gammon has been repeated. Other games that have been studied include Go [104] and Chess [122]. It is still an open question as to if and how the success of TD-Gammon can be repeated in other domains.

8.2 Robotics and Control

In recent years there have been many robotics and control applications that have used reinforcement learning. Here we will concentrate on the following four examples, although many other interesting ongoing robotics investigations are underway.

1. Schaal and Atkeson [100] constructed a two-armed robot, shown in Figure 11, that learns to juggle a device known as a devil-stick. This is a complex non-linear control task involving a six-dimensional state space and less than 200 msecs per control decision. After about 40 initial attempts the robot learns to keep juggling for hundreds of hits. A typical human learning the task requires an order of magnitude more practice to achieve proficiency at mere tens of hits.

   The juggling robot learned a world model from experience, which was generalized to unvisited states by a function approximation scheme known as locally weighted regression [25, 82].

	Training Games	Hidden Units	Results
Basic			Poor
TD 1.0	300,000	80	Lost by 13 points in 51 games
TD 2.0	800,000	40	Lost by 7 points in 38 games
TD 2.1	1,500,000	80	Lost by 1 point in 40 games

Table 2: TD-Gammon's performance in games against the top human professional players. A backgammon tournament involves playing a series of games for points until one player reaches a set target. TD-Gammon won none of these tournaments but came sufficiently close that it is now considered one of the best few players in the world.

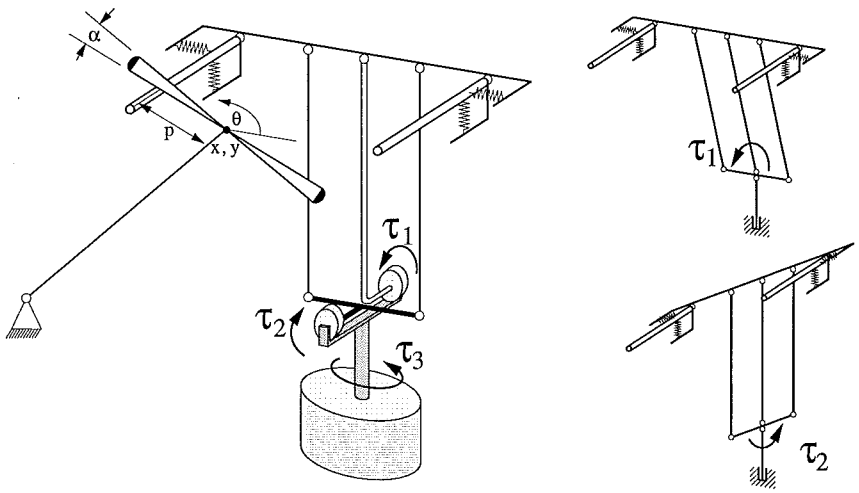

Figure 11: Schaal and Atkeson's devil-sticking robot. The tapered stick is hit alternately by each of the two hand sticks. The task is to keep the devil stick from falling for as many hits as possible. The robot has three motors indicated by torque vectors τ_1, τ_2, τ_3.

Between each trial, a form of dynamic programming specific to linear control policies and locally linear transitions was used to improve the policy. The form of dynamic programming is known as linear-quadratic-regulator design [97].

2. Mahadevan and Connell [71] discuss a task in which a mobile robot pushes large boxes for extended periods of time. Box-pushing is a well-known difficult robotics problem, characterized by immense uncertainty in the results of actions. Q-learning was used in conjunction with some novel clustering techniques designed to enable a higher-dimensional input than a tabular approach would have permitted. The robot learned to perform competitively with the performance of a human-programmed solution. Another aspect of this work, mentioned in Section 6.3, was a pre-programmed breakdown of the monolithic task description into a set of lower level tasks to be learned.

C59 C42

3. Mataric [73] describes a robotics experiment with, from the viewpoint of theoretical reinforcement learning, an unthinkably high dimensional state space, containing many dozens of degrees of freedom. Four mobile robots traveled within an enclosure collecting small disks and transporting them to a destination region. There were three enhancements to the basic Q-learning algorithm. Firstly, pre-programmed signals called *progress estimators* were used to break the monolithic task into subtasks. This was achieved in a robust manner in which the robots were not forced to use the estimators, but had the freedom to profit from the inductive bias they provided. Secondly, control was decentralized. Each robot learned its own policy independently without explicit communication with the others. Thirdly, state space was brutally quantized into a small number of discrete states according to values of a small number of pre-programmed boolean features of the underlying sensors. The performance of the Q-learned policies were almost as good as a simple hand-crafted controller for the job.

4. Q-learning has been used in an elevator dispatching task [29]. The problem, which has been implemented in simulation only at this stage, involved four elevators servicing ten floors. The objective was to minimize the average squared wait time for passengers, discounted into future time. The problem can be posed as a discrete Markov system, but there are 10^{22} states even in the most simplified version of the problem. Crites and Barto used neural networks for function approximation and provided an excellent comparison study of their Q-learning approach against the most popular and the most sophisticated elevator dispatching algorithms. The squared wait time of their controller was approximately 7% less than the best alternative algorithm ("Empty the System" heuristic with a receding horizon controller) and less than half the squared wait time of the controller most frequently used in real elevator systems.

5. The final example concerns an application of reinforcement learning by one of the authors of this survey to a packaging task from a food processing industry. The problem involves filling containers with variable numbers of non-identical products. The product characteristics also vary with time, but can be sensed. Depending on the task, various constraints are placed on the container-filling procedure. Here are three examples:

   - The mean weight of all containers produced by a shift must not be below the manufacturer's declared weight W.
   - The number of containers below the declared weight must be less than $P\%$.
   - No containers may be produced below weight W'.

   Such tasks are controlled by machinery which operates according to various *setpoints*. Conventional practice is that setpoints are chosen by human operators, but this choice is not easy as it is dependent on the current product characteristics and the current task constraints. The dependency is often difficult to model and highly non-linear. The task was posed as a finite-horizon Markov decision task in which the state of the system is a function of the product characteristics, the amount of time remaining in the production shift and the mean wastage and percent below declared in the shift so far. The system was discretized into 200,000 discrete states and local weighted regression was used to learn and generalize a transition model. Prioritized sweeping was used to maintain an optimal value function as each new piece of transition information was obtained. In simulated experiments the savings were considerable, typically with wastage reduced by a factor of ten. Since then the system has been deployed successfully in several factories within the United States.

Some interesting aspects of practical reinforcement learning come to light from these examples. The most striking is that in all cases, to make a real system work it proved necessary to supplement the fundamental algorithm with extra pre-programmed knowledge. Supplying extra knowledge

comes at a price: more human effort and insight is required and the system is subsequently less autonomous. But it is also clear that for tasks such as these, a knowledge-free approach would not have achieved worthwhile performance within the finite lifetime of the robots.

What forms did this pre-programmed knowledge take? It included an assumption of linearity for the juggling robot's policy, a manual breaking up of the task into subtasks for the two mobile-robot examples, while the box-pusher also used a clustering technique for the Q values which assumed locally consistent Q values. The four disk-collecting robots additionally used a manually discretized state space. The packaging example had far fewer dimensions and so required correspondingly weaker assumptions, but there, too, the assumption of local piecewise continuity in the transition model enabled massive reductions in the amount of learning data required.

The exploration strategies are interesting too. The juggler used careful statistical analysis to judge where to profitably experiment. However, both mobile robot applications were able to learn well with greedy exploration—always exploiting without deliberate exploration. The packaging task used optimism in the face of uncertainty. None of these strategies mirrors theoretically optimal (but computationally intractable) exploration, and yet all proved adequate.

Finally, it is also worth considering the computational regimes of these experiments. They were all very different, which indicates that the differing computational demands of various reinforcement learning algorithms do indeed have an array of differing applications. The juggler needed to make very fast decisions with low latency between each hit, but had long periods (30 seconds and more) between each trial to consolidate the experiences collected on the previous trial and to perform the more aggressive computation necessary to produce a new reactive controller on the next trial. The box-pushing robot was meant to operate autonomously for hours and so had to make decisions with a uniform length control cycle. The cycle was sufficiently long for quite substantial computations beyond simple Q-learning backups. The four disk-collecting robots were particularly interesting. Each robot had a short life of less than 20 minutes (due to battery constraints) meaning that substantial number crunching was impractical, and any significant combinatorial search would have used a significant fraction of the robot's learning lifetime. The packaging task had easy constraints. One decision was needed every few minutes. This provided opportunities for fully computing the optimal value function for the 200,000-state system between every control cycle, in addition to performing massive cross-validation-based optimization of the transition model being learned.

A great deal of further work is currently in progress on practical implementations of reinforcement learning. The insights and task constraints that they produce will have an important effect on shaping the kind of algorithms that are developed in future.

9. Conclusions

There are a variety of reinforcement-learning techniques that work effectively on a variety of small problems. But very few of these techniques scale well to larger problems. This is not because researchers have done a bad job of inventing learning techniques, but because it is very difficult to solve arbitrary problems in the general case. In order to solve highly complex problems, we must give up *tabula rasa* learning techniques and begin to incorporate bias that will give leverage to the learning process.

The necessary bias can come in a variety of forms, including the following:

shaping: The technique of shaping is used in training animals [45]; a teacher presents very simple problems to solve first, then gradually exposes the learner to more complex problems. Shaping has been used in supervised-learning systems, and can be used to train hierarchical reinforcement-learning systems from the bottom up [59], and to alleviate problems of delayed reinforcement by decreasing the delay until the problem is well understood [37, 38].

local reinforcement signals: Whenever possible, agents should be given reinforcement signals that are local. In applications in which it is possible to compute a gradient, rewarding the

agent for taking steps up the gradient, rather than just for achieving the final goal, can speed learning significantly [73].

imitation: An agent can learn by "watching" another agent perform the task [59]. For real robots, this requires perceptual abilities that are not yet available. But another strategy is to have a human supply appropriate motor commands to a robot through a joystick or steering wheel [89].

problem decomposition: Decomposing a huge learning problem into a collection of smaller ones, and providing useful reinforcement signals for the subproblems is a very powerful technique for biasing learning. Most interesting examples of robotic reinforcement learning employ this technique to some extent [28].

reflexes: One thing that keeps agents that know nothing from learning anything is that they have a hard time even finding the interesting parts of the space; they wander around at random never getting near the goal, or they are always "killed" immediately. These problems can be ameliorated by programming a set of "reflexes" that cause the agent to act initially in some way that is reasonable [73, 107]. These reflexes can eventually be overridden by more detailed and accurate learned knowledge, but they at least keep the agent alive and pointed in the right direction while it is trying to learn. Recent work by Millan [78] explores the use of reflexes to make robot learning safer and more efficient.

With appropriate biases, supplied by human programmers or teachers, complex reinforcement-learning problems will eventually be solvable. There is still much work to be done and many interesting questions remaining for learning techniques and especially regarding methods for approximating, decomposing, and incorporating bias into problems.

Acknowledgments

Thanks to Marco Dorigo and three anonymous reviewers for comments that have helped to improve this paper. Also thanks to our many colleagues in the reinforcement-learning community who have done this work and explained it to us.

Leslie Pack Kaelbling was supported in part by NSF grants IRI-9453383 and IRI-9312395. Michael Littman was supported in part by Bellcore. Andrew Moore was supported in part by an NSF Research Initiation Award and by 3M Corporation.

References

[1] David H. Ackley and Michael L. Littman. Generalization and scaling in reinforcement learning. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 550–557, San Mateo, CA, 1990. Morgan Kaufmann.

[2] J. S. Albus. A new approach to manipulator control: Cerebellar model articulation controller (cmac). *Journal of Dynamic Systems, Measurement and Control*, 97:220–227, 1975.

[3] James S. Albus. *Brains, Behavior, and Robotics.* BYTE Books, Subsidiary of McGraw-Hill, Peterborough, New Hampshire, 1981.

[4] Charles W. Anderson. *Learning and Problem Solving with Multilayer Connectionist Systems.* PhD thesis, University of Massachusetts, Amherst, MA, 1986.

[5] Rachita (Ronny) Ashar. Hierarchical learning in stochastic domains. Master's thesis, Brown University, Providence, Rhode Island, 1994.