CHAPTER 1

# Reinforcement Learning for Planning and Control

THOMAS DEAN
KEN BASYE
JOHN SHEWCHUK

## 1. Introduction

In this chapter, we consider a form of learning in which the system, referred to as the *controller*, in the course of interacting with its environment constructs a control law, referred to as a *policy*, that determines for each state enncountered by the controller an action to perform in that state. The objective is to construct an *optimal* policy, one that maximizes a given measure of performance. The system does not, however, learn by being given examples of states and the optimal actions to take in those states. Rather, the system performs actions in states and is given feedback in the form of rewards. This type of learning is called *reinforcement learning*.

Reinforcement learning is complicated by the fact that the reinforcement in the form of rewards and punishments is often intermittent and delayed. The controller may perform a long sequence of actions before receiving any reward. This makes it difficult to attribute credit or blame to actions when the reinforcement finally is received. In chess or checkers, reinforcement occurs in the form of lost pieces or lost games, and the reason for losing a piece or a game is seldom completely due to the last action taken before the loss. The problem of attributing credit or blame in such circumstances is called the *temporal credit assignment problem*, and any solution to the problems addressed in this chapter will require a solution to this problem.

The underlying problem addressed by the methods in this chapter involves making a sequence of decisions over time. The policy constructed by the controller represents a particular sort of plan that indicates the best action to take in every possible state faced by the controller. While the combinatorics involved in building and storing such a policy can be considerable, we simplify the problem by considering restricted measures of performance. Rather than measuring performance in terms of some function of sequences of states and actions of arbitrary length we employ a (possibly weighted) sum of a function states and the actions performed in those states. The result is a class of optimization problems that have been shown to be tractable in cases in which the number of states is not too large (Bellman, 1957).

## 2. Sequential Decision Making

We begin by describing a simple deterministic sequential decision making problem of the sort we would like to solve using reinforcement learning. Characterized as a decision problem, the input consists of

- a set of states, $X$, referred to as the *state space*,
- a set of actions, $U$, referred to as the *action space*,
- a set of time points, $T$,
- a *state transition function*, $f : X \times U \rightarrow X$, such that

$$x(t+1) = f(x(t), u(t)),$$

  where $x(t)$ and $u(t)$ indicate the state and action at time $t \in T$, and $t+1$ indicates the immediate successor of $t$,

- a *reward function*, $\mathrm{R} : X \times U \rightarrow \mathbf{R}$.

The output is a policy function, $\eta : X \rightarrow U$, maximizing some measure of performance. For instance, we might choose $\eta$ to maximize the *value function*, $\mathrm{V} : X \rightarrow \mathbf{R}$, where $\mathrm{V} = \mathrm{V}_n$ for some particular $n$ and $\mathrm{V}_n$ is defined by the following recurrence,

$$\mathrm{V}_i(x) = \mathrm{R}(x, \eta(x)) + \mathrm{V}_{i-1}(f(x, \eta(x)))$$

for $1 < i \leq n$, and

$$\mathrm{V}_1(x) = \mathrm{R}(x, \eta(x)).$$

The parameter, $n$, is referred to as the *lookahead* and indicates to the learning system that it need not account for the consequences of its actions beyond $n$ steps. For problems of indefinite duration in which all consequences are deemed important, we can approximate the optimal policy for unlimited lookahead by allowing $n \to \infty$.

In many of the cases we are interested in, the processes responsible for the state transitions are stochastic. For the stochastic case, instead of a state transition function, we have a set of state transition probabilities,

$$\rho_{x,x'}(u) = \Pr(x(t+1) = x' | x(t) = x, u(t) = u).$$

In describing stochastic a sequential decision making problem, we may require a somewhat more complicated reward function, $\mathrm{R} : X \times U \times X \to \mathbf{R}$, accounting for the state, the action taken in that state, and the resulting state, since the first two need not determine the third in the stochastic case.

We define the corresponding value function for $n$-step lookahead by taking expectations over the immediate rewards and next states,

$$\mathrm{V}_i(x) = \mathrm{Q}(x) + \sum_{x' \in X} \rho_{x,x'}(\eta(x))[\mathrm{V}_{i-1}(x')]$$

for $1 < i \leq n$, where

$$\mathrm{Q}(x) = \sum_{x' \in X} \rho_{x,x'}(\eta(x))\mathrm{R}(x, \eta(x), x'),$$

is referred to as the expected immediate (or *quick*) reward and $\mathrm{V}_1(x) = \mathrm{Q}(x)$.[1]

In some cases, it is unrealistic to count consequences in the distant future on an equal basis with more immediate consequences. For instance, we may mistrust our ability to make accurate long-term predictions, or future rewards may actually lose value due to some inflationary process.

1. Here we follow the notational conventions of Bellman (1957) and Howard (1960). The notation for quick rewards should not be confused with the notation used by Watkins (1989) in describing his Q-learning method. In the next section, we present Q-learning, using "W" to indicate the function that Watkins indicates with "Q."

Most biological organisms tend to discount longer-term rewards and focus on more immediate rewards. We can model this outlook on rewards by adding a discounting factor to our value function,

$$\mathrm{V}_i(x) = \mathrm{Q}(x) + \gamma \sum_{x' \in X} \rho_{x,x'}(\eta(x))[\mathrm{V}_{i-1}(x')],$$

where $0 \leq \gamma < 1$ is the *discount rate*.

Small variations in the discount rate can cause significant changes in the resulting optimal policy. For example, suppose an agent in state $x_1$ can perform one of two actions. The first action, $u_1$, with probability 1, results in a reward of $-1$ and transition to state $x_2$. The second action, $u_2$, with probability 1, results in a reward $-0.1$ and a transition back to $x_1$. Furthermore, assume that $V_\infty(x_2) = 0$. When $\gamma$ is close to zero, the agent, following the optimal policy, will perform action $u_2$ repeatedly; it will never make the transition to $x_2$. Conversely, when $\gamma$ is close to 1 the optimal policy dictates action $u_1$ resulting in a transition to $x_2$. Generally, we choose $\gamma$ close to 1 in order to maximize the agent's actual reinforcement.

Unlike the learning approaches explored elsewhere in this book, the systems considered in this chapter are not privy to a great deal of knowledge about their environment; they have access to information about the current state of the environment and are provided with feedback in the form of rewards for performing particular actions in particular states. From this state information and feedback, the system attempts to interact with its environment so as to maximize some measure of its reinforcement. Like some of the other systems considered in this book, the systems considered here compile sets of rules; for reinforcement learning the rules are of the form, if you are in state, $x$, perform action, $u$. The systems described in this chapter are designed primarily for learning how to control dynamical processes.

This chapter is not meant as a general survey of reinforcement learning methods; we are primarily interested in investigating one particular method that has enjoyed some popularity of late due to its simplicity and initial experimental success. However, a certain amount of background is required in order to understand the method and its application. In the following, we provide the essential background. A more comprehensive treatment is available in (Dean & Wellman, 1991).

## 3. Techniques

If we are given the transition probabilities and rewards, it is possible to construct the optimal policy directly without any interaction with the environment. Bellman (1957) provides an algorithm called *value iteration* that constructs an optimal policy for the $n$-step lookahead value function described earlier. Value iteration works by employing a dynamic programming approach to compute the optimal policy for $k$-step lookahead given an optimal policy for $k-1$-step lookahead. This approach provides a solution to the temporal credit assignment problem by accounting for future rewards in the assessment of prior actions.

For each $k = 1, 2, \ldots$, value iteration involves two computational steps. The first step, called *policy improvement*, involves finding the optimal policy for $k$-step lookahead by choosing actions that maximize the sum of the immediate reward and the value of the next state for the optimal policy for $k-1$-step lookahead. The second step, called *value determination*, involves computing the value function for the policy found during policy improvement. This iterative approximation scheme is guaranteed to converge to a policy that is optimal for unlimited lookahead as $k \rightarrow \infty$ (Bellman & Dreyfus, 1962). Indeed the value functions for $k = 1, 2, \ldots$ need not be computed systematically; the values for individual states can be updated in arbitrary order. As long as all of the states are updated infinitely often as the total number of updates tends to infinity, the estimated value function will converge in the limit to the value function for the optimal policy (Watkins, 1989).

There are two basic approaches to building a controller for problems in which the transition probabilities and rewards are not initially specified. In the first approach, the controller attempts to learn the transition probabilities and rewards, and then constructs an optimal policy offline using a method such as Bellman's value iteration. We call this approach the *explicit-model approach* and it is one of the standard methods used in adaptive control (Goodwin & Sin, 1984). In the second approach, the controller attempts to learn an optimal policy by constructing an evaluation function to use in selecting the best action to take when in a given state. The controller constructs this evaluation function without recourse to an explicit model for the system dynamics, and so, while the system cannot predict what the state resulting from a given action will be, it can determine whether that resulting state is better or worse than the state resulting from any other action. We call the second approach

the *direct approach*.

In the direct approach, the system can *sample* the distributions corresponding to the state transition probabilities by performing experiments in its environment. This observation suggests the following stochastic sampling version of Bellman's value iteration due to Watkins (1989). Recall that value iteration is a technique that uses successive approximation to compute a value function that converges in the limit to the value function for the optimal policy. The policy at each point in time is determined by the actions that maximize the current estimate for the optimal value function. Instead of learning a value function, stochastic value iteration involves learning what are called *action values* corresponding to the value of performing a given action in a particular state assuming that thereafter the system always chooses the action whose value is maximal. For the stochastic case with discounting, the action values should ideally be determined by

$$\mathrm{W}(x,u) = \sum_{x' \in X} \rho_{x,x'}(u)\mathrm{R}(x,u,x') + \gamma \sum_{x' \in X} \rho_{x,x'}(u)\mathrm{V}(x'),$$

where the $\eta$ used in the definition of V is

$$\eta(x) = \arg\max_u \mathrm{W}(x,u).$$

In the direct approach, however, we do not have access to the transition probabilities or rewards, and, hence, we must satisfy ourselves with an *approximation* of the function representing action values.

In this chapter, we consider several methods for approximating functions. All of these methods employ the same basic learning rule presented in the following. In general, we wish to learn some *target* function, $y$, by observing a sequence of pairs, each consisting of an *input*, $x$, and the corresponding *output*, $y(x)$. Suppose that we are trying to learn a constant function, $y(x) = C$, by observing a sequence of input/output pairs of a function corrupted by some simple noise process. For instance, let $y'(x(t)) = C + v(t)$, where $v(t)$ is a zero-mean, white gaussian noise process. The sequence of input/output pairs is defined by $\langle x(t), y'(x(t)) \rangle$ for $t = 1, 2, \ldots$. We proceed by defining a sequence of approximations, $h_1, h_2, \ldots$, defined by the following update rule,

$$h_{t+1} = h_t + \beta_t \epsilon_t,$$

where the *error* term, $\epsilon_t$, is defined as

$$\epsilon_t = y'(x(t)) - h_t,$$

and $\beta_t$ determines the learning rate of the update rule. For $\beta_t = \frac{1}{t}$, $h_t$ is guaranteed to converge to the value $C$ as $t \to \infty$. This update rule is called the *least mean square (LMS) rule* and is due to Widrow and Hoff (1960). Given a set of *features* corresponding to arbitrary functions of the input, the LMS rule can be used to approximate any function corresponding to a linear combination of these features.

Function approximation is somewhat more complicated in the case of reinforcement learning in that the training data does not include the output of the target function or even the noise-corrupted output of the target function as in the example above. Instead the controller must continually revise an estimate of the target function by sampling the rewards provided in the course of interaction with the environment.

We can adapt the LMS rule to learning the value function for a fixed policy as follows,

$$\mathrm{V}_{t+1}(x(t)) = \mathrm{V}_t(x(t)) + \beta_t \epsilon_t,$$

where in this case,

$$\epsilon_t = [r(t+1) + \gamma \mathrm{V}_t(x(t+1))] - \mathrm{V}_t(x(t)),$$

and $r(t)$ is the reward at time $t$. The quantity enclosed in the square brackets is an estimate based on the latest sampled reward. In lieu of the state transition probabilities, we make use of the only information we have on hand in computing the error, namely the immediate reward and next state. The rule for learning action values is quite similar,

$$\begin{aligned} \mathrm{W}_{t+1}(x(t), u(t)) &= \mathrm{W}_t(x(t), u(t)) + \\ &\quad \beta_t[r(t+1) + \gamma \mathrm{V}_t(x(t+1)) - \mathrm{W}_t(x(t), u(t))], \end{aligned}$$

where in this case,

$$\mathrm{V}_t(x) = \max_u \mathrm{W}_t(x, u).$$

In order to use action values to learn an optimal policy, we need an exploration strategy for improving policies and learning action values. If the controller always follows the best action as determined by the current action values, then it will likely have an accurate estimate of the action values for a particular policy, but it will not likely find the optimal policy simply because by not deviating from the current best actions it may never even try the optimal policy. On the other hand,

```
Procedure SVI(β, ξ)
   x  =  Random(X)
   do forever
      ρ  =  Random({z|0 ≤ z ≤ 1})
      if ρ  ≤  ξ
         then u  =  arg max_u W(x, u)
         else u  =  Random(U)
      ⟨x', r⟩  =  Simulate(x, u)
      W(x, u)  =  W(x, u) + β[r + γ max_v W(x', v) − W(x, u)]
      x  =  x'
```

*Figure 1–1.* Stochastic value iteration

if the controller is always experimenting with alternative policies, then the action values will never converge to an accurate estimate.

Figure 1–1 shows a simple algorithm, SVI, for stochastic value iteration. The subroutine, Simulate$(x, u)$, takes a state and an action and returns the next state and reward in accord with the state transition probabilities and reward function. The subroutine, Random$(s)$, returns a (pseudo) random element from the set $s$. The algorithm chooses the current best action—as determined by the action values—a fixed percentage, $\xi$, of the time, and chooses a random action the rest of the time. The quantity, $1 - \xi$, is the percentage of time that the controller spends performing exploratory actions.

The SVI algorithm is appropriate for simple problems involving a small number of discrete states and actions. For problems involving larger state and action spaces, learning can be painfully slow without some mechanism for compactly representing and generalizing action values.

In its simplest form, we are attempting to learn W, a mapping from $X \times U \rightarrow \mathbf{R}$. If $X$ and $U$ are continuous and W is piecewise *smooth* (i.e., small changes in $X$ and $U$ produce correspondingly small changes in $\mathbf{R}$), then we can employ techniques from regularization theory (Tikhonov, 1963; Poggio & Girosi, 1989) that exploit smoothness in order to facilitate generalization. In the following, we describe one such technique that employs hashing to map a large space into a significantly smaller one.

In the hashing technique, the input space of the target function is partitioned into a set of regions that are mapped onto the smaller space

using one or more hashing functions. The smaller space is represented by a finite number of storage elements containing the parameters for a piecewise-constant approximation. Learning proceeds by adjusting these parameters using the LMS rule. The method is generally referred to as the *CMAC* approach, after the name given to it by Albus, the Cerebellar Model Articulation Controller (Albus, 1975).

We consider two different CMAC methods. The first we refer to as *vanilla CMAC*. Vanilla CMAC employs multiple partitions of the input space that are offset from one another to form overlapping regions. Each of the partitions is mapped onto the set of storage elements with a different hashing function. For a given input, the output of vanilla CMAC is the average of contents of the storage elements determined by hashing the set of regions containing the input.

The second CMAC method is called *multiresolution CMAC* (Moody, 1989). This method employs a successive refinement strategy to achieve a useful tradeoff between the speed and the accuracy of learning. Suppose you wish to learn a function; call it $y_1$. To do so you construct a CMAC system in which the partitions consist of regions that are rather large. This CMAC system will find an approximation to $y_1$, call it $f_1$, very quickly, but the approximation is likely to be a poor one, given the coarseness of the mapping. To correct for the inaccuracies of $f_1$, we build another CMAC system to learn the function, $y_2 = y_1 - f_1$, but this system makes use of partitions consisting of somewhat smaller regions. This second CMAC system will find an approximation to $y_2$, call it $f_2$, more slowly than the first CMAC, but it will still represent $y_2$ more accurately than $f_1$ represented $y_1$, and the sum of the two functions, $f_1 + f_2$, will be a better approximation of $y_1$ than $f_1$ alone. We can continue in this manner to define a sequence of CMAC systems, each system using a finer partition than the one before it in the sequence, and each providing a correction for the function corresponding to the sum of functions provided by the CMAC systems occurring earlier in the sequence.

One way to implement the above strategy is for the learning system to apply each CMAC system in stages, starting with the system using the coarsest partitions and proceeding to those using finer partitions. Each CMAC is run for a fixed number of steps using a learning schedule that tends to zero. This sequential implementation has the disadvantage that it cannot adapt if the target function changes slowly over time. An

alternative implementation is to run all of the CMAC systems in parallel, using a different fixed learning rate for each CMAC such that the finer the partition the slower the learning (the smaller the fixed rate) (Shewchuk & Dean, 1990). This parallel approach tends to learn somewhat slower than the sequential approach, but the parallel approach is still quite fast and its ability to adapt to handle time-varying functions makes it useful in a number of applications for which the staged approach would not be effective.

At this point, we have all of the basic tools in place for building effective reinforcement learning algorithms. Before we proceed with our empirical investigations, we consider some additional complications that arise in reinforcement learning.

## 4. Complications

The revised value estimates used in computing the error terms in learning a value function or action values are just a special case of estimating expected long-term returns on the basis of some number of observed rewards (Sutton, 1988). In general, we can make use of any number of observed rewards using estimates of the form,

$$\begin{aligned} \mathrm{V}_t(x_{t-1}) &= r(t) + \gamma r(t+1) + \gamma^2 r(t+2) + \cdots + \\ &\quad \gamma^{n-1} r(t+n-1) + \gamma^n \max_u \mathrm{W}_t(x(t+n), u). \end{aligned}$$

Estimates involving several observed rewards can speed the assignment of credit in cases of delayed reinforcement. In the deterministic case, estimates with more observations are generally better in that they provide more accurate estimates and speed learning, but they also require more memory and computation. The stochastic case is more complicated. If there is a great deal of variance in the actual returns, then using more observations will likely provide a poor estimate of the expected returns. However, if the corrections provided by $\mathrm{W}_t$ are significantly in error, then the estimates will be biased, and using more observations will serve to reduce the effect of this bias.

In addition to the increased memory and computation and the problems introduced by stochastic processes, using estimates with more observations is further complicated by the fact that subsequent rewards are assumed to be due to following some fixed policy. In order to find

the optimal policy, the controller must periodically deviate from its current policy. As a very simple method of accounting for deviations from the current policy, we can introduce an additional term, $\lambda_t$, equal to zero if the action taken at $t$ deviates from the current policy and equal to one otherwise,

$$\begin{aligned} \mathrm{V}_t(x_{t-1}) &= r(t) + \lambda_{t+1}[\gamma r(t+1) + \cdots + \\ &\quad \lambda_{t+n-1}[\gamma^{n-1} r(t+n-1) + \\ &\quad \lambda_{t+n}[\gamma^n \max_u \mathrm{W}_t(x(t+n), u)]] \ldots]. \end{aligned}$$

By choosing values for $\lambda_t$ intermediate between zero and one for actions dictated by the current policy, it is possible to obtain reasonable tradeoffs between bias and variance for the problems noted in the previous paragraph. By choosing a value of $\lambda_t$ that reflects some measure of the deviation of the action performed from the action dictated, it is possible to gain some of the advantages of using multiple observed rewards in cases in which all or most of the actions performed by the controller deviate somewhat from the actions dictated by the current policy (Watkins, 1989).

There are other credit assignment problems besides temporal credit assignment that require a solution in order to build effective reinforcement learning systems. In many reinforcement learning problems, there are significant subspaces of the state space that are either unimportant or for which the optimal response is the same throughout. In the case of a large, high-dimensional state space, it would be impractical to attempt to represent in memory such a space at a high level of precision. The *structural credit assignment problem* involves apportioning available memory according to the perceived distribution of inputs and the variability of the target function. In some cases, apportionment can be facilitated by choosing an appropriate representation for states and action. For instance, if a sensor provides only two bits of relevant information, then it makes sense to represent it as boolean and not as an integer. This not only saves on storage, but can also speed learning by requiring less training data.

Even given a well-designed representation for the states and actions, there can be significant problems brought on by the high dimensionality of the state and action spaces. A robot control systems typically involves dozens of sensors and effectors; even if they can all be represented as boolean variables learning in a space of size $2^{100}$ can be a

daunting task. To deal with the problems raised by large state and action spaces, researchers have become interested in methods for compressing large spaces and decomposing large high-dimensional spaces into a number of smaller low-dimensional spaces. CMAC is an example of a method that employs compression to economize on space and facilitate generalization.

Dimensionality reduction provides a more powerful approach to dealing with structural credit assignment. It is seldom the case in a well-designed representation that any of the dimensions of the corresponding space are entirely irrelevant. However, it is quite often the case that for a small subspace only a few of the dimensions need to be allocated storage for learning action values or storing a policy. Methods that dynamically create and combine subspaces to optimize the use of available storage are just beginning to be explored in the literature; in the final section of this chapter, we provide some relevant pointers. Before leaving this section, we point out two additional problems that plague the methods described in this chapter.

Problems arise in implementations of stochastic value iteration—such as the `SVI` algorithm—that interleave the acquisition of action values with policy improvement. If the policy is revised too frequently, then the estimates for the action values fail to converge to the correct values, and policy improvement fails. On the other hand, if the policy is revised infrequently, learning can proceed quite slowly. Determining an effective exploration strategy can be quite difficult in practice. Systems that actively direct their exploratory activities on the basis of their experience can gain a distinct advantage over systems that employ simple random exploration strategies (Cohn *et al.*, 1990), but we will not consider such methods in this chapter.

Complex tasks requiring the execution of a long sequence of actions prior to receiving any reward are particularly difficult for the methods described in this chapter. The problem is that the action values typically have to be represented with high precision in order for the system to choose an optimal policy for performing such tasks. However, if you are trying to learn quickly, you generally set the learning rates high and use low-resolution encodings of the state and action spaces, both of which have the effect of sacrificing precision. As with interleaving value determination and policy improvement, we do not have simple solution to the problem of aquiring accurate action values for long sequences

| | | $\rho_{ij}(u)$ | | | $\mathrm{R}(i,u,j)$ | | |
|---|---|---|---|---|---|---|---|
| $i$ | $u$ | $j=$ 1 | 2 | 3 | $j=$ 1 | 2 | 3 |
| 1 | $a$ | 1/2 | 1/4 | 1/4 | 10 | 4 | 8 |
| | $b$ | 1/16 | 3/4 | 3/16 | 8 | 2 | 4 |
| | $c$ | 1/4 | 1/8 | 5/8 | 4 | 6 | 4 |
| 2 | $a$ | 1/2 | 0 | 1/2 | 14 | 0 | 18 |
| | $b$ | 1/16 | 7/8 | 1/16 | 8 | 16 | 8 |
| 3 | $a$ | 1/4 | 1/4 | 1/2 | 10 | 2 | 8 |
| | $b$ | 1/8 | 3/4 | 1/8 | 6 | 4 | 2 |
| | $c$ | 3/4 | 1/16 | 3/16 | 4 | 0 | 8 |

*Table 1–1.* Specification for a simple stochastic process

of actions; it is just another problem that one has to be aware of in applying the methods described in this chapter.

## 5. Experiments

In the following, we consider a number of experiments that illustrate various aspects of reinforcement learning. The first experiment illustrates the effect of varying amounts of exploration on convergence. The experiment involves a stochastic process described in (Howard, 1960), consisting of three states, $\{1,2,3\}$, three actions, $\{a,b,c\}$, and a discount rate, $\gamma$, of 0.4. The state transition probabilities and rewards are shown in Table 1–1, where $\mathrm{R}(i,u,j)$ indicates the reward on performing action, $u$, starting in state, $i$, and ending in state, $j$. For this experiment, we use the stochastic value iteration algorithm, `SVI`, shown in Figure 1–1, and we invoke `SVI` with $\alpha = 0.1$, and $\xi$ varying between 0 and 1.

Our measure of performance for this experiment is the time, $t$, at which the current policy,

$$\eta_t(x) = \arg\max_u \mathrm{W}_t(x,u),$$

first adheres to the same policy for at least 500 iterations of the simulator. We refer to this time as the *policy stabilization time*, and to the

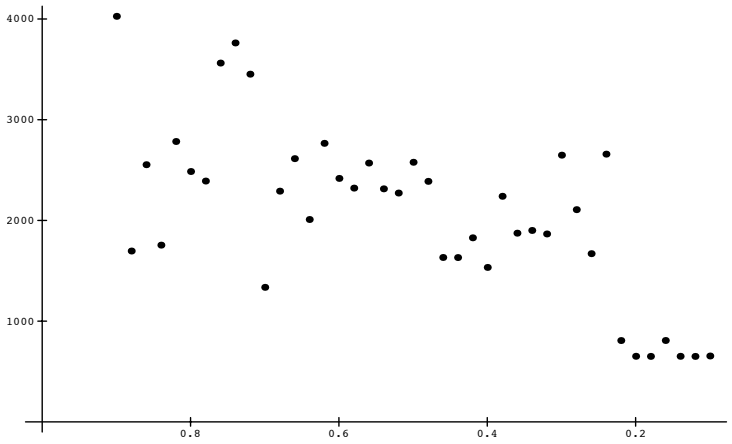


*Figure 1–2.* Policy stabilization time as a function of the percentage of time spent in performing exploratory actions

particular policy as the *earliest stable policy*.

Figure 1–2 shows a graph of policy stabilization time as a function of the percentage of time, $1 - \xi$, that the controller spends in performing exploratory actions. For $\xi < 0.7$ the earliest stable policy is the optimal policy, but for $\xi \geq 0.7$ it is not.[2] Since the system always deviates from the current policy some percentage of the time, there is always the chance that even after acquiring and then remaining with the optimal policy for a significant interval of time the system will temporarily switch to some suboptimal policy.

If the percentage of exploration, $1 - \xi$, is small enough ($< 0.3$) then the system will appear to converge to a policy which is suboptimal even though in the limit as $t \to \infty$ the system will spend the major portion of its time with optimal policy. Maintaining a high percentage of exploration may speed early policy stabilization but the controller will perform poorly as the majority of its actions are not dictated by its optimal policy but rather by its random exploratory actions. The tendency to deviate from the optimal can be eliminated by gradually reducing the percentage of exploration to zero. While exploration strategies can be tuned for particular applications, for the problems we have looked at, an exploration rate of around 15% works well.

2. The graph in Figure 1–2 was compiled using the average of several trials for each value of $\xi$.

```
Procedure SVIC(β, ξ)
   x  =  x_0
   do forever
      ρ  =  Random({z|0 ≤ z ≤ 1})
      if ρ  ≤  ξ
         then k  =  arg max_i CMAC_i(x)
         else k  =  Random({1, 2, 3})
      ⟨x', r⟩  =  Simulate(x, k)
      w_k  :=  w_k + β[r + γ max_i CMAC_i(x') − CMAC_k(x)]φ_k(x)
      x  =  x'
```

*Figure 1–3.* Learning algorithm for the inverted-pendulum problem

In the next experiment, we use vanilla CMAC and stochastic value iteration to learn to balance an inverted pendulum. We simulate the dynamical system using the equations of motion for the inverted-pendulum problem described in (Barto *et al.*, 1983). We follow Barto *et al.* (1983) in measuring performance using the length of time the system can keep the pendulum in an upright position. We call this the *time-til-failure* performance measure. In this experiment, the only reinforcement that the system receives is when the pendulum attains an angle of $\pm 12^\circ$, at which point the reinforcement is some fixed negative value.

Learning is broken up into a sequence of *trials*. In each trial, the pendulum is set randomly to $\pm 0.1^\circ$, and training continues until the pendulum attains an angle of $\pm 12^\circ$.

We restrict the space of possible actions by using a *bang-bang* control scheme in which there are only three possible actions: impulse left, corresponding to a force of $+20$ newtons; impulse right, corresponding to a force of $-20$ newtons; and no impulse, corresponding to a force of 0 newtons. This control scheme considerably simplifies the problem of encoding actions. The sample period for the controller is 0.02 seconds.

The learning algorithm that we use in this experiment is a variant of stochastic value iteration. Figure 1–3 lists the procedure, `SVIC`, for stochastic value iteration using CMAC. `SVIC` employs three CMAC systems, one for learning the action values for each of the three actions. Let the integers 1, 2, and 3 designate the three actions, impulse left, impulse right, and no impulse, respectively, and $\mathrm{CMAC}_1$, $\mathrm{CMAC}_2$, and $\mathrm{CMAC}_3$ designate the corresponding CMAC systems. If $x$ is a state (i.e., a three-dimensional vector encoding the angular position and ve-

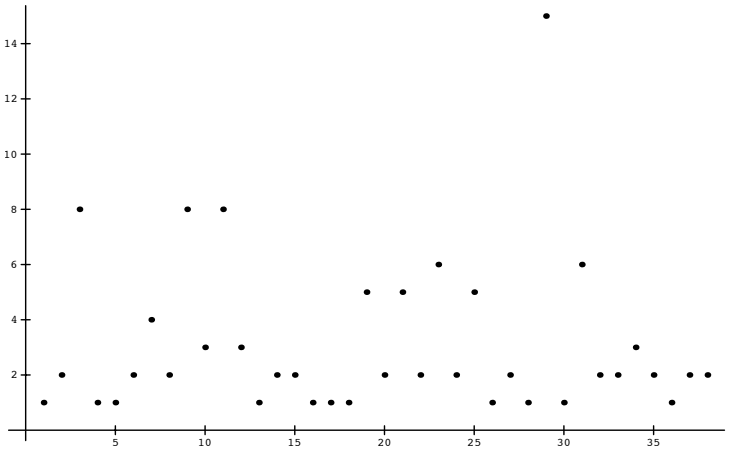


*Figure 1–4.* Performance of SVIC on the inverted-pendulum problem

locity of the pendulum, and the horizontal velocity of the cart) then $\mathrm{CMAC}_i(x)$ is used to estimate the value of performing action $i$ in state $x$. The initial state is denoted $x_0$, and the percentage of time that the system performs exploratory actions, $1-\xi$, is 0.15. Let $\mathbf{w}_i$ and $\boldsymbol{\phi}_i$ denote the parameter vector and mapping functions for $\mathrm{CMAC}_i$ (see Dean and Wellman (1991) for additional CMAC implementation details). Each CMAC uses 64 storage elements.

Figure 1–4 shows a graph of time-til-failure in seconds as a function of the number of trials. For the first 38 trials, the time-til-failure is less than 10 seconds except at the 29th trial at which point the system manages to keep the pole upright for about 16 seconds. The 39th trial lasted for over ten minutes of simulated time before we terminated the run with the pendulum in an upright position and the deviations from vertical apparently converging to zero. Recall that it is only upon failure that the system receives a nonzero reinforcement. The system was able to learn to balance the pendulum with only 38 failures.

In related experiments, the system is given continuous reinforcement so that the reward at each point in time is inversely proportional to the absolute value of the angle that the pendulum makes with the vertical. In the experiments with continuous reinforcement, the controller was able to learn to balance the pendulum in just a few seconds of simulated time. In the next and last set of experiments, the controller is given continuous negative reinforcement except when it attains a particular goal

configuration at which point it receives a large positive reinforcement.

## 6. Related Work

For additional background material, Dean and Wellman (1991) provide a more detailed introduction to the issues involved in stochastic sequential decision making, function approximation, and reinforcement learning. Barto *et al.* (1983) also provide an overview of research on reinforcement learning and sequential decision making.

A number of approaches have been proposed to deal with the problems of generalization and structural credit assignment in high-dimensional state spaces. Chapman and Kaelbling (1991) propose a method for recursively splitting the state space on the basis of statistical measures of the reinforcement received. Their method is similar to that employed in classification for inducing decision trees (Breiman *et al.*, 1984; Quinlan, 1986) with the important difference that in reinforcement learning the controller is not provided with state/action pairs but rather with sporadic reinforcement data; the reinforcement data requires more sophisticated statistial tests to guide recursive splitting. Mahadevan and Connell (1991) employ a related method that instead of starting with complete space and splitting it, starts with a completely fragmented space and proceeds to merge fragments. Shewchuk has developed methods for partitioning the state and action spaces to optimize the use of available storage based on variants of $k$-means and nearest-neighbor clustering techniques. All of these reinforcement methods are closely related to the unsupervised learning methods used in pattern recognition for statistical clustering (Duda and Hart, 1973).

Even relatively simple tasks (e.g., the task faced by a mobile robot in entering a particular room and docking with a charging unit) can require a training period that would challenge the most patient of teachers. If you are supplying the rewards, one solution is to decompose the task into component subtasks (e.g., follow the wall on your right until you notice a door, enter the room, align with the charging unit) and reward the system for performing each subtask individually (Lin, 1991; Mahadevan & Connell, 1991). After the system is competent at performing each of the subtasks, you have to program the system to perform the appropriate sequence of subtasks in the absence of intermediate (subtask) rewards. Training is somewhat more complicated, but often requires

less time overall.

Researchers have attempted to speed learning by a variety of means. Sutton (1990) considers some of the issues involved in acquiring and then using a predictive model to supplement the experience acquired through direct interaction with the environment. Lin (1991) describes a scheme for saving and then replaying interaction sequences multiple times to consolidate learning. Whitehead (1991) shows that, in some cases, a controller can considerably reduce the time necessary to acquire an optimal policy if it can observe the behavior of another controller that has already acquired the optimal policy.

## References

James S. Albus. A new approach to manipulator control: The cerebellar model articulation controller (CMAC). *Journal of Dynamic Systems, Measurement, and Control*, 97:270–277, 1975.

Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13(5):835–846, 1983.

Andrew G. Barto, Richard S. Sutton, and Christopher J. C. H. Watkins. Learning and sequential decision making. In Michael Gabriel and John Moore, editors, *Learning and Computational Neuroscience: Foundations of Adaptive Networks*. MIT Press, Cambridge, Massachusetts, 1990.

Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

Richard Bellman and Stuart Dreyfus. *Applied Dynamic Programming*. Princeton University Press, Princeton, New Jersey, 1962.

L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, California, 1984.

David Chapman and Leslie Kaelbling. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings IJCAI 12*. IJCAII, 1991.

David Cohn, Les Atlas, Richard Ladner, M. A. El-Sharkawi, R. J.

Marks, M.E. Aggoune, and D. C. Park. Training connectionist networks with queries and selective sampling. In *Neural Information Processing Systems*. Morgan Kaufmann, 1990.

Thomas Dean and Michael Wellman. *Planning and Control*. Morgan Kaufmann, San Mateo, California, 1991.

R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis*. John Wiley and Sons, New York, 1973.

G. C. Goodwin and K. S. Sin. *Adaptive Filtering Prediction and Control*. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

Ronald A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, Massachusetts, 1960.

Long-Ji Lin. Programming robots using reinforcement learning and teaching. In *Proceedings AAAI-91*. AAAI, 1991.

Sridhar Mahadevan and Jonathan Connell. Automatic programming of behavior-based robots using reinforcement learning. In *Proceedings AAAI-91*. AAAI, 1991.

John E. Moody. Fast learning in multi-resolution hierarchies. In David Touretsky, editor, *Advances in Neural Information Processing*. Morgan-Kaufmann, Los Altos, California, 1989.

Tomaso Poggio and Federico Girosi. A theory of networks for approximation and learning. Technical Report AI Memo No. 1140, MIT AI Laboratory, 1989.

J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

John Shewchuk. *Reinforcement Learning in Dynamical Systems With High Input and Output Dimensionality*. PhD thesis, Brown University, Providence, RI, Forthcoming.

John Shewchuk and Thomas Dean. Towards learning time-varying functions with high input dimensionality. In *Proceedings of the Fifth IEEE International Symposium on Intelligent Control*, pages 383–388. IEEE, 1990.

Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

Richard S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings 7th International Conference on Machine Learning*, 1990.

A. N. Tikhonov. Solution of incorrectly formulated problems and the regularization method. *Soviet Math. Dokl.*, 4:1035–1038, 1963.

C. J. C. H Watkins. *Learning from Delayed Rewards.* PhD thesis, Cambridge University, 1989.

Steven D. Whitehead. A complexity analysis of cooperative mechanisms in reinforcement learning. In *Proceedings AAAI-91.* AAAI, 1991.

B. Widrow and M. E. Hoff. Adaptive switching circuits. In *1960 WESCON Convention Record Part IV, (Reprinted in J. A. Anderson and E. Rosenfeld,* Neurocomputing: Foundations of Research, *The MIT Press, Cambridge, MA, 1988)*, pages 96–104, 1960.