A Tutorial Survey of Reinforcement Learning

S SATHIYA KEERTHI and B RAVINDRAN
Department of Computer Science and Automation
Indian Institute of Science, Bangalore
e-mail: {ssk,ravi}@chanakya.csa.iisc.ernet.in

Abstract. This paper gives a compact, self–contained tutorial survey of reinforcement learning, a tool that is increasingly finding application in the development of intelligent dynamic systems. Research on reinforcement learning during the past decade has led to the development of a variety of useful algorithms. This paper surveys the literature and presents the algorithms in a cohesive framework.

Keywords. Reinforcement learning, dynamic programming, optimal control, neural networks

1 Introduction

Reinforcement Learning (RL), a term borrowed from animal learning literature by Minsky (1954, 1961), refers to a class of learning tasks and algorithms in which the learning system learns an associative mapping, $\pi : X \rightarrow A$ by maximizing a scalar evaluation (reinforcement) of its performance from the environment (user). Compared to supervised learning, in which for each x shown the environment provides the learning system with the value of $\pi(x)$, RL is more difficult since it has to work with much less feedback from the environment. If, at some time, given an $x \in X$, the learning system tries an $a \in A$ and, the environment immediately returns a scalar reinforcement evaluation of the (x, a) pair (that indicates how far a is from $\pi(x)$) then we are faced with an *immediate* RL task. A more difficult RL task is *delayed* RL, in which the environment only gives a single scalar reinforcement evaluation, collectively for $\{(x_t, a_t)\}$, a sequence of (x, a) pairs occuring in time during the system operation. Delayed RL tasks commonly arise in optimal control of dynamic systems and planning problems of AI. In this paper our main interest is in the solution of delayed RL problems. However, we also study immediate RL problems because methods of solving them play an useful role in the solution of delayed RL problems.

Delayed RL encompasses a diverse collection of ideas having roots in animal learning (Barto 1985; Sutton & Barto 1987), control theory (Bertsekas 1989; Kumar 1985), and AI (Dean & Wellman 1991). Delayed RL algorithms were first employed by Samuel (1959, 1967) in his celebrated work on playing checkers. However, it

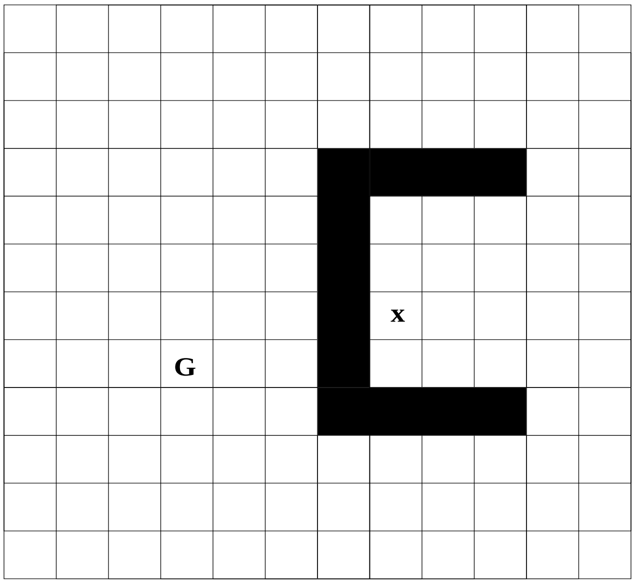

Figure 1. Navigating in a grid world.

was only much later, after the publication of Barto, Sutton and Anderson's work (Barto *et al* 1983) on a delayed RL algorithm called *adaptive heuristic critic* and its application to the control problem of pole balancing, that research on RL got off to a flying start. Watkins' Q-Learning algorithm (Watkins 1989) made another impact on the research. A number of significant ideas have rapidly emerged during the past five years and the field has reached a certain level of maturity. In this paper we provide a comprehensive tutorial survey of various ideas and methods of delayed RL. To avoid distractions and unnecessary clutter of notations, we present all ideas in an intuitive, not-so-rigorous fashion. In preparing this tutorial, we have obtained a lot of guidance from the works of Watkins (1989), Barto, Sutton and Watkins (1990), Barto, Bradtke and Singh (1992), Bradtke (1994), and Barto (1992).

To illustrate the key features of a delayed RL task let us consider a simple example.

Example 1 Navigating a Robot

Figure 1 illustrates a grid world in which a robot navigates. Each blank cell on the grid is called a *state*. Shaded cells represent barriers; these are not states. Let X be the state space, i.e., the set of states. The cell marked G is the goal state. The aim is to reach G from any state in the least number of time steps. Navigation is done using four *actions*: $A = \{N, S, E, W\}$, the actions denoting the four possible movements along the coordinate directions.

Rules of transition are defined as follows. Suppose that the robot is in state x and action N is chosen. Then the resulting next state, y is the state directly to the north of x, *if there is such a state*; otherwise $y = x$. For instance, choosing W at the x shown in figure 1 will lead to the system staying at x. The goal state is a special case. By definition we will take it that any action taken from the goal state results in a transition back to the goal state. In more general problems, the rules of transition can be stochastic.

The robot moves at discrete (integer) time points starting from $t = 0$. At a time step t, when the robot is at state, x_t, we define an immediate reward[1] as

$$r(x_t) = \begin{cases} 0 & \text{if} \quad x_t = G, \\ -1 & \text{otherwise.} \end{cases}$$

In effect, the robot is penalized for every time step spent at non-goal states. It is simple to verify that maximizing the *total reward* over time,

$$V(x) = \sum_{t=0}^{\infty} r(x_t)$$

is equivalent to achieving minimum time navigation from the starting state, $x_0 = x$. Let $V^\star(x)$ denote the maximum achievable (optimal) value of $V(x)$.

We are interested in finding a feedback policy, $\pi : X \rightarrow A$ such that, if we start from any starting state and select actions using π then we will always reach the goal in the minimum number of time steps.

The usefulness of immediate RL methods in delayed RL can be roughly explained as follows. Typical delayed RL methods maintain $\hat{V}$, an approximation of the optimal function, $V^\star$. If action a is performed at state x and state y results, then $\hat{V}(y)$ can be taken as an (approximate) immediate evaluation of the (x, a) pair.[2] By solving an immediate RL problem that uses this evaluation function we can obtain a good sub–optimal policy for the delayed RL problem. We present relevant immediate RL algorithms in §2.

□

Delayed RL problems are much harder to solve than immediate RL problems for the following reason. Suppose, in example 1, performance of a sequence of actions, selected according to some policy, leads the robot to the goal. To improve the policy using the experience, we need to evaluate the goodness of each action performed. But the total reward obtained gives only the cumulative effect of all actions performed. Some scheme must be found to reasonably apportion the cumulative evaluation to the individual actions. This is referred to as the *temporal credit assignment problem.* (In the previous paragraph we have already given a hint of how delayed RL methods do temporal credit assignment.)

Dynamic programming (DP) (Bertsekas 1989; Ross 1983) is a well–known tool for solving problems such as the one in example 1. It is an off–line method that requires the availability of a complete model of the environment. But the concerns of delayed RL are very different. To see th clearly let us return to example 1 and impose the requirement that the robot has no knowledge of the environment and that the only way of learning is by on–line experience of trying various actions[3] and thereby visiting many states. Delayed RL algorithms are particularly meant for such situations and have the following general format.

Delayed RL Algorithm

Initialize the learning system.

Repeat

---

[1] Sometimes r is referred to as the primary reinforcement. In more general situations, r is a function of x_t as well as a_t, the action at time step t.

[2] An optimal action at x is one that gives the maximum value of $V^\star(y)$.

[3] During learning this is usually achieved by using a (stochastic) exploration policy for choosing actions. Typically the exploration policy is chosen to be totally random at the beginning of learning and made to approach an optimal policy as learning nears completion.

1. *With the system at state x, choose an action a according to an exploration policy and apply it to the system.*
2. *The environment returns a reward, r, and also yields the next state, y.*
3. *Use the experience, (x, a, r, y) to update the learning system.*
4. *Set $x := y$.*

Even when a model of the environment is available, it is often advantageous to avoid an off–line method such as DP and instead use a delayed RL algorithm. This is because, in many problems the state space is very large; while a DP algorithm operates with the entire state space, a delayed RL algorithm only operates on parts of the state space that are most relevant to the system operation. When a model is available, delayed RL algorithms can employ simulation mode of operation instead of on–line operation so as to speed–up learning and avoid doing experiments using hardware. In this paper, we will use the term, *real time operation* to mean that either on–line operation or simulation mode of operation is used.

In most applications, representing functions such as $V^\star$ and π exactly is infeasible. A better alternative is to employ parametric function approximators, e.g., connectionist networks. Such approximators must be suitably chosen for use in a delayed RL algorithm. To clarify this, let us take $V^\star$ for instance and consider a function approximator, $\hat{V}(\cdot; w) : X \rightarrow R$, for it. Here R denotes the real line and w denotes the vector of parameters of the approximator that is to be learnt so that $\hat{V}$ approximates $V^\star$ well. Usually, at step 3 of the delayed RL algorithm, the learning system uses the experience to come up with a direction, η in which $\hat{V}(x; w)$ has to be changed for improving performance. Given a step size, β, the function approximator must alter w to a new value, w^{new} so that

$$\hat{V}(x; w^{\text{new}}) = \hat{V}(x; w) + \beta\eta \tag{1}$$

For example, in multilayer perceptrons (Hertz *et al* 1991) w denotes the set of weights and thresholds in the network and, their updating can be carried out using backpropagation so as to achieve (1). In the rest of the paper we will denote the updating process in (1) as

$$\hat{V}(x; w) := \hat{V}(x; w) + \beta\eta \tag{2}$$

and refer to it as a *learning rule*.

The paper is organized as follows. Section 2 discusses immediate RL. In §3 we formulate Delayed RL problems and mention some basic results. Methods of estimating total reward are discussed in §4. These methods play an important role in delayed RL algorithms. DP techniques and delayed RL algorithms are presented in §5. Section 6 addresses various practical issues. We make a few concluding remarks in §7.

2 Immediate Reinforcement Learning

Immediate RL refers to the learning of an associative mapping, $\pi : X \rightarrow A$ given a reinforcement evaluator. To learn, the learning system interacts in a closed loop with the environment. At each time step, the environment chooses an $x \in X$ and, the learning system uses its function approximator, $\hat{\pi}(\cdot; w)$ to select an action: $a = \hat{\pi}(x; w)$. Based on both x and a, the environment returns an evaluation or "reinforcement", $r(x, a) \in R$. Ideally, the learning system has to adjust w so as to produce the maximum possible r value for each x; in other words, we would like $\hat{\pi}$ to solve the parametric global optimization problem,

$$r(x, \hat{\pi}(x; w)) = r^{\star}(x) \stackrel{\text{def}}{=} \max_{a \in A} r(x, a) \quad \forall x \in X \tag{3}$$

Supervised learning is a popular paradigm for learning associative mappings (Hertz *et al* 1991). In supervised learning, for each x shown the supervisor provides the learning system with the value of $\pi(x)$. Immediate RL and supervised learning differ in the following two important ways.

- In supervised learning, when an x is shown and the supervisor provides $a = \pi(x)$, the learning system forms the directed information, $\eta = a - \hat{\pi}(x; w)$ and uses the learning rule: $\hat{\pi}(x; w) := \hat{\pi}(x; w) + \alpha\eta$, where α is a (positive) step size. For immediate RL such directed information in not available and so it has to employ some strategy to obtain such information.
- In supervised learning, the learning system can simply check if $\eta = 0$ and hence decide whether the correct map value has been formed by $\hat{\pi}$ at x. However, in immediate RL, such a conclusion on correctness cannot be made without exploring the values of $r(x, a)$ for all a.

Therefore, immediate RL problems are much more difficult to solve than supervised learning problems.

A number of immediate RL algorithms have been described in the literature. Stochastic learning automata algorithms (Narendra & Thathachar 1989) deal with the special case in which X is a singleton, A is a finite set, and $r \in [0, 1]$. The Associative Reward-Penalty (A_{R-P}) algorithm (Barto & Anandan 1985; Barto *et al* 1985; Barto & Jordan 1987; Mazzoni *et al* 1990) extends the learning automata ideas to the case where X is a finite set. Williams (1986, 1987) has proposed a class of immediate RL methods and has presented interesting theoretical results. Gullapalli (1990, 1992a) has developed algorithms for the general case in which X, A are finite-dimensional real spaces and r is real valued. Here we will discuss only algorithms which are most relevant to, and useful in delayed RL.

One simple way of solving (3) is to take one x at a time, use a global optimization algorithm (e.g., complete enumeration) to explore the A space and obtain the correct a for the given x, and then make the function approximator learn this (x, a) pair. However, such an idea is not used for the following reason. In most situations where immediate RL is used as a tool (e.g., to approximate a policy in delayed RL), the learning system has little control over the choice of x. When, at a given x, the learning system chooses a particular a and sends it to the environment for evaluation, the environment not only sends a reinforcement evaluation but also alters the x value. Immediate RL seeks approaches which are appropriate to these situations.

Let us first consider the case in which A is a finite set: $A = \{a^1, a^2, \cdots, a^m\}$. Let R^m denote the m–dimensional real space. The function approximator, $\hat{\pi}$ is usually formed as a composition of two functions: a function approximator, $g(\cdot; w) : X \rightarrow R^m$

and a fixed function, $M : R^m \rightarrow A$. The idea behind this set-up is as follows. For each given x, $z = g(x; w) \in R^m$ gives a vector of merits of the various a^i values. Let z_k denote the k–th component of z. Given the merit vector z, $a = M(z)$ is formed by the max selector,

$$a = a^k \quad \text{where} \quad z_k = \max_{1 \leq i \leq m} z_i \tag{4}$$

Let us now come to the issue of learning (i.e., choosing a w). At some stage, let x be the input, z be the merit vector returned by g, and a^k be the action having the largest merit value. The environment returns the reinforcement, $r(x, a^k)$. In order to learn we need to evaluate the goodness of z^k (and therefore, the goodness of a^k). Obviously, we cannot do this using existing information. We need an estimator, call it $\hat{r}(x; v)$, that provides an estimate of $r^\star(x)$. The difference, $r(x, a^k) - \hat{r}(x; v)$ is a measure of the goodness of a^k. Then a simple learning rule is

$$g_k(x; w) := g_k(x; w) + \alpha(r(x, a^k) - \hat{r}(x; v)) \tag{5}$$

where α is a small (positive) step size.

Learning $\hat{r}$ requires that all members of A are evaluated by the environment at each x. Clearly, the max selector, (4) is not suitable for such exploration. For instance, if at some stage of learning, for some x, g assigns the largest merit to a wrong action, say a^k, and $\hat{r}$ gives, by mistake, a value smaller than $r(x, a^k)$, then no action other than a^k is going to be generated by the learning system at the given x. So we replace (4) by a controlled stochastic action selector that generates actions randomly when learning begins and approaches (4) as learning is completed. A popular stochastic action selector is based on the Boltzmann distribution,

$$p_i(x) \stackrel{\text{def}}{=} \text{Prob}\{a = a^i | x\} = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)} \tag{6}$$

where T is a nonnegative real parameter (temperature) that controls the stochasticity of the action selector. For a given x the expected reinforcement of the action selector is

$$\tilde{r}(x) \stackrel{\text{def}}{=} \text{E}(r(x, a)|x) = \sum_i p_i(x) r(x, a^i)$$

As $T \rightarrow 0$ the stochastic action selector approaches the max selector, (4), and, $\tilde{r}(x) \rightarrow r^\star(x)$. Therefore we train $\hat{r}$ to approximate $\tilde{r}$ (instead of $r^\star$). This is easy to do because, for any fixed value of T, $\tilde{r}$ can be estimated by the average of the performance of the stochastic action selector over time. A simple learning rule that achieves this is

$$\hat{r}(x; v) := \hat{r}(x; v) + \beta(r(x, a) - \hat{r}(x; v)) \tag{7}$$

where β is a small (positive) step size.

Remark Two important comments should be made regarding the convergence of learning rules such as (7) (we will come across many such learning rules later) which are designed to estimate an expectation by averaging over time.

- Even if $\hat{r} \equiv \tilde{r}$, $r(x, a) - \hat{r}(x; v)$ can be non-zero and even large in size. This is because a is only an instance generated by the distribution, $p(x)$. Therefore, to avoid unlearning as $\hat{r}$ comes close to $\tilde{r}$, the step size, β must be controlled properly. The value of β may be chosen to be 1 when learning begins, and then slowly decreased to 0 as learning progreses.

- For good learning to take place, the sequence of x values at which (7) is carried out must be such that it covers all parts of the space, X as often as possible. Of course, when the learning system has no control over the choice of x, it can do nothing to achieve such an exploration. To explore, the following is usually done. Learning is done over a number of *trials*. A trial consists of beginning with a random choice of x and operating the system for several time steps. At any one time step, the system is at some x and the learning system chooses an action, a and learns using (7). Depending on x, a and the rules of the environment a new x results and the next time step begins. Usually, when learning is repeated over multiple trials, the X space is thoroughly explored.

Let us now consider the case in which A is continuous, say a finite dimensional real space. The idea of using merit values is not suitable. It is better to directly deal with a function approximator, $h(\cdot; w)$ from X to A. In order to do exploration a controlled random perturbation, η is added to $h(x;w)$ to form $a = \hat{\pi}(x)$. A simple choice is to take η to be a Gaussian with zero mean and having a standard deviation, $\sigma(T)$ that satisfies: $\sigma(T) \rightarrow 0$ as $T \rightarrow 0$. The setting-up and training of the reinforcement estimator, $\hat{r}$ is as in the case when A is discrete. The function approximator, h can adopt the following learning rule:

$$h(x;w) := h(x;w) + \alpha(r(x,a) - \hat{r}(x;v))\eta \tag{8}$$

where α is a small (positive) step size. In problems where a bound on $r^\star$ is available, this bound can be suitably employed to guide exploration, i.e., choose σ (Gullapalli 1990).

Jordan and Rumelhart (1990) have suggested a method of 'forward models' for continuous action spaces. If r is a known differentiable function, then a simple, deterministic learning law based on gradient ascent can be given to update $\hat{\pi}$:

$$\hat{\pi}(x;w) := \hat{\pi}(x;w) + \alpha \frac{\partial r(x,a)}{\partial a} \tag{9}$$

If r is not known, Jordan and Rumelhart suggest that it is learnt using on–line data, and (9) be used using this learnt r. If for a given x, the function $r(x,\cdot)$ has local maxima then the $\hat{\pi}(x)$ obtained using learning rule, (9) may not converge to $\pi(x)$. Typically this is not a serious problem. The stochastic approach discussed earlier does not suffer from local maxima problems. However, we should add that, because the deterministic method explores in systematic directions and the stochastic method explores in random directions, the former is expected to be much faster. The comparison is very similar to the comparison of deterministic and stochastic techniques of continuous optimization.

3 Delayed Reinforcement Learning

Delayed RL concerns the solution of stochastic optimal control problems. In this section we discuss the basics of such problems. Solution methods for delayed RL will be presented in §4 and §5. In these three sections we will mainly consider problems in which the state and control spaces are finite sets. This is because the main issues and solution methods of delayed RL can be easily explained for such problems. We will deal with continuous state and/or action spaces briefly in §5.

Consider a discrete-time stochastic dynamic system with a finite set of states, X. Let the system begin its operation at $t = 0$. At time t the *agent (controller)* observes state[4] x_t and, selects (and performs) action a_t from a finite set, $A(x_t)$, of possible actions. Assume that the system is Markovian and stationary, i.e.,

$$\begin{aligned}\text{Prob}\{x_{t+1} = y \mid & \; x_0, a_0, x_1, a_1, \cdots, x_t = x, a_t = a\} \\ & = \text{Prob}\{x_{t+1} = y | x_t = x, a_t = a\} \;\; \stackrel{\text{def}}{=} \;\; P_{xy}(a)\end{aligned}$$

A *policy* is a method adopted by the agent to choose actions. The objective of the decision task is to find a policy that is optimal according to a well defined sense, described below. In general, the action specified by the agent's policy at some time can depend on the entire past history of the system. Here we restrict attention to policies that specify actions based only on the current state of the system. A deterministic policy, π defines, for each $x \in X$ an action $\pi(x) \in A(x)$. A stochastic policy, π defines, for each $x \in X$ a probability distribution on the set of feasible actions at x, i.e., it gives the values of $\text{Prob}\{\pi(x) = a\}$ for all $a \in A(x)$. For the sake of keeping the notations simple we consider only deterministic policies in this section. All ideas can be easily extended to stochastic policies using appropriate detailed notations.

Let us now precisely define the optimality criterion. While at state x, if the agent performs action a, it receives an immediate *payoff* or *reward*, $r(x, a)$. Given a policy π we define the *value function*, $V^\pi : X \rightarrow R$ as follows:

$$V^\pi(x) = \text{E}\{\sum_{t=0}^{\infty} \gamma^t r(x_t, \pi(x_t)) | x_0 = x\} \tag{10}$$

Here future rewards are discounted by a factor $\gamma \in [0, 1)$. The case $\gamma = 1$ is avoided only because it leads to some difficulties associated with the existence of the summation in (10). Of course, these difficulties can be handled by putting appropriate assumptions on the problem solved. But, to avoid unnecessary distraction we do not go into the details; see (Bradtke 1994; Bertsekas & Tsitsiklis 1989).

The expectation in (10) should be understood as

$$V^\pi(x) = \lim_{N \rightarrow \infty} \;\; \text{E}\{\sum_{t=0}^{N-1} \gamma^t r(x_t, \pi(x_t)) | x_0 = x\}$$

where the probability with which a particular state sequence, $\{x_t\}_{t=0}^{N-1}$ occurs is taken in an obvious way using $x_0 = x$ and repeatedly employing π and P. We wish to maximize the value function:

$$V^\star(x) = \max_\pi V^\pi(x) \quad \forall x \tag{11}$$

$V^\star$ is referred to as the optimal value function. Because $0 \leq \gamma < 1$, $V^\pi(x)$ is bounded. Also, since the number of π's is finite $V^\star(x)$ exists.

[4] If the state is not completely observable then a method that uses the observable states and retains past information has to be used; see (Bacharach 1991; Bacharach 1992; Chrisman 1992; Mozer & Bacharach 1990a, 1990b; Whitehead and Ballard 1990).

How do we define an optimal policy, $\pi^\star$? For a given x let $\pi^{x,\star}$ denote a policy that achieves the maximum in (11). Thus we have a collection of policies, $\{\pi^{x,\star} : x \in X\}$. Now $\pi^\star$ is defined by picking only the first action from each of these policies:

$$\pi^\star(x) = \pi^{x,\star}(x) \ , \quad x \in X$$

It turns out that $\pi^\star$ achieves the maximum in (11) for every $x \in X$. In other words,

$$V^\star(x) = V^{\pi^\star}(x) \ , \quad x \in X \tag{12}$$

This result is easy to see if one looks at Bellman's optimality equation – an important equation that $V^\star$ satisfies:

$$V^\star(x) = \max_{a \in A(x)} \left[r(x,a) + \gamma \sum_{y \in X} P_{xy}(a) V^\star(y) \right] \tag{13}$$

The fact that $V^\star$ satisfies (13) can be explained as follows. The term within square brackets on the right hand side is the total reward that one would get if action a is chosen at the first time step and then the system performs optimally in all future time steps. Clearly, this term cannot exceed $V^\star(x)$ since that would violate the definition of $V^\star(x)$ in (11); also, if $a = \pi^{x,\star}(x)$ then this term should equal $V^\star(x)$. Thus (13) holds. It also turns out that $V^\star$ is the unique function from X to R that satisfies (13) for all $x \in X$. This fact, however, requires a non-trivial proof; details can be found in (Ross 1983; Bertsekas 1989; Bertsekas & Tsitsiklis 1989).

The above discussion also yields a mechanism for computing $\pi^\star$ if $V^\star$ is known:

$$\pi^\star(x) = \arg \max_{a \in A(x)} \left[r(x,a) + \gamma \sum_{y \in X} P_{xy}(a) V^\star(y) \right]$$

A difficulty with this computation is that the system model, i.e., the function, $P_{xy}(a)$ must be known. This difficulty can be overcome if, instead of the V–function we employ another function called the Q-function. Let $\mathcal{U} = \{(x,a) : x \in X, a \in A(x)\}$, the set of feasible (state,action) pairs. For a given policy π, let us define $Q^\pi : \mathcal{U} \rightarrow R$ by

$$Q^\pi(x,a) = r(x,a) + \gamma \sum_{y \in X} P_{xy}(a) V^\pi(y) \tag{14}$$

Thus $Q^\pi(x,a)$ denotes the total reward obtained by choosing a as the first action and then following π for all future time steps. Let $Q^\star = Q^{\pi^\star}$. By Bellman's optimality equation and (12) we get

$$V^\star(x) = \max_{a \in A(x)} [Q^\star(x,a)] \tag{15}$$

It is also useful to rewrite Bellman's optimality equation using $Q^\star$ alone:

$$Q^\star(x,a) = r(x,a) + \gamma \sum_{y \in X} P_{xy}(a) \{ \max_{b \in A(y)} Q^\star(y,b) \} \tag{16}$$

Using $Q^\star$ we can compute $\pi^\star$:

$$\pi^\star(x) = \arg \max_{a \in A(x)} [Q^\star(x,a)] \tag{17}$$

Thus, if $Q^\star$ is known then $\pi^\star$ can be computed without using a system model. This advantage of the Q–function over the V–function will play a crucial role in §5 for deriving a model–free delayed RL algorithm called Q–Learning (Watkins 1989).

Let us now consider a few examples that give useful hints for problem formulation. These examples are also commonly mentioned in the RL literature.

Example 2 Navigating a Robot with Dynamics

In example 1 the robot is moved from one cell to another like the way pieces are moved in a chess board. True robot motions, however, involve dynamics; the effects of velocity and acceleration need to be considered. In this example we will include dynamics in a crude way, one that is appropriate to the grid world. Let h_t and v_t denote the horizontal and vertical coordinates of the cell occupied by the robot at time t, and, $\dot{h}_t$ and $\dot{v}_t$ denote the velocities. The vector, $(h_t, v_t, \dot{h}_t, \dot{v}_t)$ denotes the system state at time t; each one of the four components is an integer. The goal state is $x^G = (h^G, v^G, 0, 0)$ where (h^G, v^G) is the coordinate vector of the goal cell G. In other words, the robot has to come to rest at G. Let $\dot{h}_{\max}$ and $\dot{v}_{\max}$ be limits on velocity magnitudes. Thus the state space is given by

$$\tilde{X} = \{x = (h, v, \dot{h}, \dot{v}) | \quad (h, v) \text{ is a blank cell, } |\dot{h}| \leq \dot{h}_{\max}, \text{ and } |\dot{v}| \leq \dot{v}_{\max}\}$$

We will also include an extra state, f called failure state to denote situations where a barrier (shaded) cell is entered or a velocity limit is exceeded. Thus

$$X = \tilde{X} \cup \{f\}$$

The accelerations [5] along the horizontal and vertical directions, respectively a^h and a^v, are the actions. To keep h and v as integers let us assume that each of the accelerations takes only even integer values. Let $a_{\max}$ be a positive even integer that denotes the limit on the magnitude of accelerations. Thus $a = (a^h, a^v)$ is an admissible action if each of a^h and a^v is an even integer lying in $[-a_{\max}, a_{\max}]$.

As in example 1 state transitions are deterministic. They are defined as follows. If barrier cells and velocity limits are not present, then application of action (a^h, a^v) at $x_t = (h_t, v_t, \dot{h}_t, \dot{v}_t)$ will lead to the next state $x'_{t+1} = (h'_{t+1}, v'_{t+1}, \dot{h}'_{t+1}, \dot{v}'_{t+1})$ given by

$$\begin{array}{ll} h'_{t+1} = h_t + \dot{h}_t + a^h/2 \ , & v'_{t+1} = v_t + \dot{v}_t + a^v/2 \\ \dot{h}'_{t+1} = \dot{h}_t + a^h \ , & \dot{v}'_{t+1} = \dot{v}_t + a^v \end{array}$$

Let $\mathcal{C}$ denote the curve in the grids world resulting during the transition from (h_t, v_t) at time t to (h'_{t+1}, v'_{t+1}) at time $(t+1)$, i.e., the solution of the differential equations: $d^2h/d\tau^2 = a^h$, $d^2v/d\tau^2 = a^v$, $\tau \in [t, t+1]$, $h(t) = h_t$, $dh/d\tau|_\tau = \dot{h}_t$, $v(t) = v_t$, $dv/d\tau|_\tau = \dot{v}_t$. If, either $\mathcal{C}$ cuts across a barrier cell or $(\dot{h}'_{t+1}, \dot{v}'_{t+1})$ is an inadmissible velocity vector, then we say failure has occured during transition. Thus state transitions are defined as

$$x_{t+1} = \begin{cases} f & \text{if } x_t = f \\ f & \text{if failure occurs during transition} \\ x^G & \text{if } x^t = x^G \\ x'_{t+1} & \text{otherwise} \end{cases}$$

[5] Negative acceleration will mean deceleration.

The primary aim is to avoid failure. Next, among all failure-avoiding trajectories we would like to choose the trajectory which reaches the goal state, $x^G = (h^G, v^G, 0, 0)$ in as few time steps as possible. These aims are met if we define

$$r(x,a) = \begin{cases} -1 & \text{if} \quad x = f, \\ 1 & \text{if} \quad x = x^G, \\ 0 & \text{otherwise.} \end{cases}$$

The following can be easily checked.

- $V^\star(x) < 0$ iff there does not exist a trajectory starting from x that avoids failure.
- $V^\star(x) = 0$ iff, starting from x, there exists a failure-avoiding trajectory, but there does not exist a trajectory that reaches G.
- $V^\star(x) > 0$ iff, starting from x, there exists a failure-avoiding trajectory that also reaches G; also, an optimal policy $\pi^\star$ leads to the generation of a trajectory that reaches G in the fewest number of steps from x while avoiding failure.

□

Example 3 Playing Backgammon
Consider a game of backgammon (Magriel 1976) between players A and B. Let us look at the game from A's perspective, assuming that B follows a fixed policy. Now A can make a decision on a move only when the current board pattern as well as its dice roll are known. Therefore a state consists of a (board pattern, dice roll) pair. Each action consists of a set of marker movements. State transition is defined as follows.

- A moves its markers in accordance with the chosen action. This step is deterministic, and results in a new board pattern.
- B rolls the dice. This step is stochastic.
- B moves its markers according to its policy. This step can be deterministic or stochastic depending on the type of B's policy.
- A rolls the dice. This step is stochastic.

The set of states that correspond to A's win is the set of goal states, G to be reached. We can define the reward as: $r(x,a) = 1$ if x is a goal state; and $r(x,a) = 0$ otherwise. If $\gamma = 1$, then for a given policy, say π, the value function $V^\pi(x)$ will denote the probability that A will win from that state.

□

Example 4 Pole Balancing
We now deviate from our problem formulation and present an example that involves continuous state/action spaces. A standard problem for learning controllers is that of balancing an inverted pendulum pivoted on a trolley, a problem similar to that of balancing a stick on one's hand (Barto *et al* 1983). The system comprises a straight horizontal track, like a railway track, with a carriage free to move along it. On the carriage is an axis, perpendicular to the track and pointing out to the side, about which a pendulum is free to turn. The controller's task is to keep the

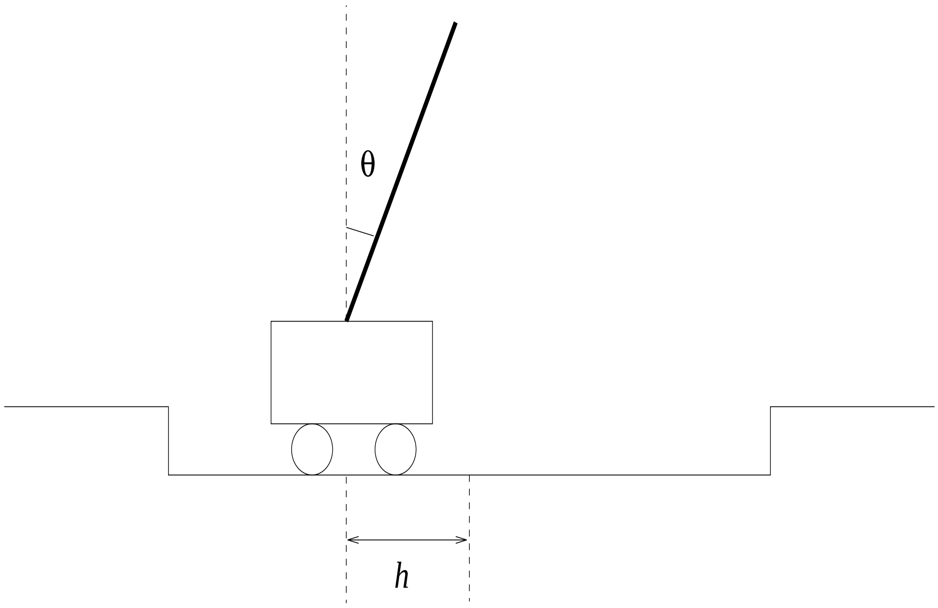

Figure 2. Pole balancing.

pendulum upright, by alternately pulling and pushing the carriage along the track. Let h and θ be as shown in figure 2. We say balancing has failed if anyone of the following inequalities is violated:

$$h \leq h_{\max}, \quad h \geq -h_{\max}, \quad \theta \leq \theta_{\max}, \quad \theta \geq -\theta_{\max}$$

where $h_{\max}$ and $\theta_{\max}$ are specified bounds on the magnitudes of h and θ. The aim is to balance without failure for as long a time as possible.

The state of the system is the 4–tuple, $(h, \dot{h}, \theta, \dot{\theta})$, where $\dot{h}$ and $\dot{\theta}$ are the time derivatives of h and θ respectively. The action is the force applied to the carriage. It takes real values in the interval, $[-F_{\max}, F_{\max}]$. To simplify the problem solution, sometimes the action space is taken to be $\{-F_{\max}, F_{\max}\}$ (Michie & Chambers 1968; Barto *et al* 1983; Anderson 1989). A discrete time formulation of the problem is obtained by cutting continuous time (non-negative real line) into uniform time intervals, each of duration Δ, and taking the applied force to be constant within each interval.[6] The state of the system at the continuous time instant, $t\Delta$ is taken to be x_t, the discrete time state at the t–th time step. The mechanical dynamics of the system defines state transition, except for one change: once failure occurs, we will assume, for the sake of consistent problem formulation, that the system stays at failure for ever.

As in example 2 we will take the state space to be $X = \tilde{X} \cup \{f\}$, where

$$\tilde{X} = \{x = (h, \dot{h}, \theta, \dot{\theta}) | -h_{\max} \leq h \leq h_{\max}, \quad -\theta_{\max} \leq \theta \leq \theta_{\max}\}$$

and f is the failure state that collectively represents all states not in $\tilde{X}$. Since the

[6] This constant is the action for the time step corresponding to that interval.

aim is to avoid failure, we choose

$$r(x,a) = \begin{cases} -1 & \text{if} \quad x = f, \\ 0 & \text{otherwise.} \end{cases}$$

□

4 Methods of Estimating V^π and Q^π

Delayed RL methods use a knowledge of V^π (Q^π) in two crucial ways: (1) the optimality of π can be checked by seeing if V^π (Q^π) satisfies Bellman's optimality equation; and (2) if π is not optimal then V^π (Q^π) can be used to improve π. We will elaborate on these details in the next section. In this section we discuss, in some detail, methods of estimating V^π for a given policy, π. (Methods of estimating Q^π are similar and so we will deal with them briefly at the end of the section.) Our aim is to find $\hat{V}(\cdot; v)$, a function approximator that estimates V^π. Much of the material in this section is taken from the works of Watkins (1989), Sutton (1984, 1988) and Jaakkola *et al* (1994).

To avoid clumsiness we employ some simplifying notations. Since π is fixed we will omit the superscript from V^π and so call it as V. We will refer to $r(x_t, \pi(x_t))$ simply as r_t. If p is a random variable, we will use p to denote both, the random variable as well as an instance of the random variable.

A simple approximation of $V(x)$ is the *n–step truncated return*,

$$V^{[n]}(x) = \sum_{\tau=0}^{n-1} \gamma^\tau r_\tau \ , \quad \hat{V}(x; v) = \mathrm{E}(V^{[n]}(x)) \tag{18}$$

(Here it is understood that $x_0 = x$. Thus, throughout this section τ will denote the number of time steps elapsed after the system passed through state x. It is for stressing this point that we have used τ instead of t. In a given situation, the use of time – is it 'actual system time' or 'time relative to the occurence of x' – will be obvious from the context.) If $r_{\max}$ is a bound on the size of r then it is easy to verify that

$$\max_x |\hat{V}(x; v) - V(x)| \leq \frac{\gamma^n r_{\max}}{(1-\gamma)} \tag{19}$$

Thus, as $n \to \infty$, $\hat{V}(x; v)$ converges to $V(x)$ uniformly in x.

But (18) suffers from an important drawback. The computation of the expectation requires the complete enumeration of the probability tree of all possible states reachable in n time steps. Since the breadth of this tree may grow very large with n, the computations can become very burdensome. One way of avoiding this problem is to set

$$\hat{V}(x; v) = V^{[n]}(x) \tag{20}$$

where $V^{[n]}(x)$ is obtained via either Monte-Carlo simulation or experiments on the real system (the latter choice is the only way to systems for which a model is unavailable.) The approximation, (20) suffers from a different drawback. Because the breadth of the probability tree grows with n, the variance of $V^{[n]}(x)$ also grows with n. Thus $\hat{V}(x; v)$ in (20) will not be a good approximation of $E(V^{[n]}(x))$ unless it is obtained as an average over a large number of trials.[7] Averaging is achieved if we use a learning rule (similar to (7)):

$$\hat{V}(x;v) := \hat{V}(x;v) + \beta\left[V^{[n]}(x) - \hat{V}(x;v)\right] \tag{21}$$

where $\beta \in (0,1)$ is a small step size. Learning can begin with a random choice of v. Eventually, after a number of trials, we expect the $\hat{V}$ resulting from (21) to satisfy (19).

In the above approach, an approximation of V, $\hat{V}$ is always available. Therefore, an estimate that is *more appropriate than* $V^{[n]}(x)$ is the *corrected n–step truncated return,*

$$V^{(n)}(x) = \sum_{\tau=0}^{n-1} \gamma^{\tau} r_{\tau} + \gamma^{n} \hat{V}(x_n;v) \tag{22}$$

where x_n is the state that occurs n time steps after the system passed through state x. Let us do some analysis to justify this statement.

First, consider the ideal learning rule,

$$\hat{V}(x;v) := E(V^{(n)}(x)) \quad \forall\, x \tag{23}$$

Suppose v gets modified to v_{new} in the process of satisfying (23). Then, similar to (19) we can easily derive

$$\max_x |\hat{V}(x;v_{\text{new}}) - V(x)| \le \gamma^n \max_x |\hat{V}(x;v) - V(x)|$$

Thus, as we go through a number of learning steps we achieve $\hat{V} \to V$. Note that this convergence is achieved even if n is fixed at a small value, say $n = 1$. On the other hand, for a fixed n, the learning rule based on $V^{[n]}$, i.e., (18), is only guaranteed to achieve the bound in (19). *Therefore, when a system model is available it is best to choose a small n, say n = 1, and employ (23).*

Now suppose that, either a model is unavailable or (23) is to be avoided because it is expensive. In this case, a suitable learning rule that employs $V^{(n)}$ and uses real–time data is:

$$\hat{V}(x;v) := \hat{V}(x;v) + \beta\left[V^{(n)}(x) - \hat{V}(x;v)\right] \tag{24}$$

Which is better: (21) or (24)? There are two reasons as to why (24) is better.

- Suppose $\hat{V}$ is a good estimate of V. Then a small n makes $V^{(n)}$ ideal: $V^{(n)}(x)$ has a mean close to $V(x)$ and it also has a small variance. Small variance means that (24) will lead to fast averaging and hence fast convergence of $\hat{V}$ to V. On the other hand n has to be chosen large for $V^{[n]}(x)$ to have a mean close to $V(x)$; but then, $V^{[n]}(x)$ will have a large variance and (21) will lead to slow averaging.
- If $\hat{V}$ is not a good estimate of V then both $V^{(n)}$ and $V^{[n]}$ will require a large n for their means to be good. If a large n is used, the difference between $V^{(n)}$ and $V^{[n]}$, i.e., $\gamma^n \hat{V}$ is negligible and so both (21) and (24) will yield similar performance.

[7] As already mentioned, a trial consists of starting the system at a random state and then running the system for a number of time steps.

The above discussion implies that it is better to employ $V^{(n)}$ than $V^{[n]}$. It is also clear that, when $V^{(n)}$ is used, a suitable value of n has to be chosen dynamically according to the goodness of $\hat{V}$. To aid the manipulation of n, Sutton (1988) suggested a new estimate constructed by geometrically averaging $\{V^{(n)}(x) : n \geq 1\}$:

$$V^{\lambda}(x) = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} V^{(n)}(x) \tag{25}$$

Here $(1-\lambda)$ is a normalizing term. Sutton referred to the learning algorithm that uses V^{λ} as $TD(\lambda)$. Here TD stands for 'Temporal Difference'. The use of this name will be justified below. Expanding (25) using (22) we get

$$\begin{aligned} V^{\lambda}(x) &= (1-\lambda)\left[V^{(1)}(x) + \lambda V^{(2)}(x) + \lambda^2 V^{(3)}(x) + \cdots\right] \\ &= r_0 + \gamma(1-\lambda)\hat{V}(x_1; v) \ + \\ &\quad \gamma\lambda \Big[r_1 + \gamma(1-\lambda)\hat{V}(x_2; v) \ + \\ &\quad\quad \gamma\lambda \Big[r_2 + \gamma(1-\lambda)\hat{V}(x_3; v) \ + \\ &\quad\quad\quad \cdots \end{aligned} \tag{26}$$

Using the fact that $r_0 = r(x, \pi(x))$ the above expression may be rewritten recursively as

$$V^{\lambda}(x) = r(x, \pi(x)) + \gamma(1-\lambda)\hat{V}(x_1; v) + \gamma\lambda V^{\lambda}(x_1) \tag{27}$$

where x_1 is the state occuring a time step after x. Putting $\lambda = 0$ gives $V^0 = V^{(1)}$ and putting $\lambda = 1$ gives $V^1 = V$, which is the same as $V^{(\infty)}$. Thus, the range of values obtained using $V^{(n)}$ and varying n from 1 to ∞ is approximately achieved by using V^{λ} and varying λ from 0 to 1. *A simple idea is to use V^{λ} instead of $V^{(n)}$, begin the learning process with $\lambda = 1$, and reduce λ towards zero as learning progresses and $\hat{V}$ becomes a better estimate of V.* If λ is properly chosen then a significant betterment of computational efficiency is usually achieved when compared to simply using $\lambda = 0$ or $\lambda = 1$ (Sutton 1988). In a recent paper, Sutton and Singh (1994) have developed automatic schemes for doing this assuming that no cycles are present in state trajectories.

The definition of V^{λ} involves all $V^{(n)}$s and so it appears that we have to wait for ever to compute it. However, computations involving V^{λ} can be nicely rearranged and then suitably approximated to yield a practical algorithm *that is suited for doing learning concurrently with real time system operation.* Consider the learning rule in which we use V^{λ} instead of $V^{(n)}$:

$$\hat{V}(x; v) := \hat{V}(x; v) + \beta \left[V^{\lambda}(x) - \hat{V}(x; v)\right] \tag{28}$$

Define the *temporal difference* operator, Δ by

$$\Delta(x) = r(x, \pi(x)) + \gamma\hat{V}(x_1; v) - \hat{V}(x; v) \tag{29}$$

$\Delta(x)$ is the difference of predictions (of $V^{\pi}(x)$) at two consecutive time steps: $r(x, \pi(x)) + \gamma\hat{V}(x_1; v)$ is a prediction based on information at $\tau = 1$, and $\hat{V}(x; v)$ is a prediction based on information at $\tau = 0$. Hence the name, 'temporal difference'. Note that $\Delta(x)$ can be easily computed using the experience within a time step. A simple rearrangement of the terms in the second line of (26) yields

$$V^{\lambda}(x) - \hat{V}(x; v) = \Delta(x) + (\gamma\lambda)\Delta(x_1) + (\gamma\lambda)^2\Delta(x_2) + \cdots \tag{30}$$

Even (30) is not in a form suitable for use in (28) because it involves future terms, $\Delta(x_1)$, $\Delta(x_2)$, etc., extending to infinite time. One way to handle this problem is to choose a large N, accumulate $\Delta(x)$, $\Delta(x_1)$, $\cdots$, $\Delta(x_{N-1})$ in memory, truncate the right hand side of (30) to include only the first N terms, and apply (28) at $\tau = N+1$, i.e., $(N+1)$ time steps after x occured. However, a simpler and approximate way of achieving (30) is to include the effects of the temporal differences as and when they occur in time. Let us say that the system is in state x at time t. When the systems transits to state x_1 at time $(t+1)$, compute $\Delta(x)$ and update $\hat{V}$ according to: $\hat{V}(x;v) := \hat{V}(x;v) + \beta(\gamma\lambda)\Delta(x_1)$. When the system transits to state x_2 at time $(t+2)$, compute $\Delta(x_1)$ and update $\hat{V}$ according to: $\hat{V}(x;v) := \hat{V}(x;v)+\beta(\gamma\lambda)^2\Delta(x_2)$ and so on. The reason why this is approximate is because $\hat{V}(x;v)$ is continuously altered in this process whereas (30) uses the $\hat{V}(x;v)$ existing at time t. However, if β is small and so $\hat{V}(x;v)$ is adapted slowly, the approximate updating method is expected to be close to (28).

One way of implementing the above idea is to maintain an *eligibility trace*, $e(x,t)$, for each state visited (Klopf 1972; Klopf 1982; Klopf 1988; Barto *et al* 1983; Watkins 1989), and use the following learning rule at time t:

$$\hat{V}(x;v) := \hat{V}(x;v) + \beta e(x,t)\Delta(x_t) \ \ \forall\ x \tag{31}$$

where x_t is the system state at time t. The eligibility traces can be adapted according to

$$e(x,t) = \begin{cases} 0 & \text{if } x \text{ has never been visited} \\ \gamma\lambda e(x,t-1) & \text{if } x_t \neq x \\ 1+\gamma\lambda e(x,t-1) & \text{if } x_t = x \end{cases} \tag{32}$$

Two important remarks must be made regarding this implementation scheme.

- Whereas the previous learning rules (e.g., (21), (24) and (28)) update $\hat{V}$ only for one x at a time step, (31) updates the $\hat{V}$ of all states with positive eligibility trace, at a time step. Rule (31) is suitable for connectionist implementation, but not so for implementations on sequential computers. A more efficient way is to keep track of the last k states visited and update $\hat{V}$ for them only. The value of k should depend on λ. If λ is small, k should be small. If $\lambda = 0$ then $k = 1$.

- The rule for updating eligibility traces, (32) assumes that learning takes place in a single trial. If learning is done over multiple trials then all eligibility traces must be reset to zero just before each new trial is begun.

The remark made below equation (7) applies as well to the learning rules, (21), (24), (28) and, (31). Dayan and Sejnowski (1993), and Jaakkola *et al* (1994) have shown that, if the real time $TD(\lambda)$ learning rule, (31) is used, then under appropriate assumptions on the variation of β in time, as $t \rightarrow \infty$, $\hat{V}$ converges to V^π with probability one. Practically, learning can be achieved by doing multiple trials and decreasing β towards zero as learning progresses.

Thus far in this section we have assumed that the policy, π is deterministic. If π is a stochastic policy then all the ideas of this section still hold with appropriate interpretations: all expectations should include the stochasticity of π, and, the $\pi(x)$ used in (27), (29) etc. should be taken as instances generated by the stochastic policy.

Let us now come to the estimation of Q^π. Recall from (14) that $Q^\pi(x,a)$ denotes the total reward obtained by choosing a as the first action and then following π for all future time steps. Details concerning the extension of Q^π are clearly described in a recent report by Rummery and Niranjan (1994). Let $\hat{Q}(x,a;v)$ be the estimator of $Q^\pi(x,a)$ that is to be learnt concurrently with real time system operation. Following the same lines of argument as used for the value function, we obtain a learning rule similar to (31):

$$\hat{Q}(x,a;v) := \hat{Q}(x,a;v) + \beta e_Q(x,a,t)\Delta_Q(x_t,a_t) \ \ \forall\ (x,a) \tag{33}$$

where: x_t and a_t are, respectively, the system state and the action chosen at time t;

$$\Delta_Q(x,a) = r(x,a) + \gamma\hat{Q}(x_1,\pi(x_1);v) - \hat{Q}(x,a;v); \tag{34}$$

and

$$e_Q(x,a,t) = \begin{cases} 0 & \text{if } (x,a) \text{ has never been visited} \\ \gamma\lambda e_Q(x,a,t-1) & \text{if } (x_t,a_t) \neq (x,a) \\ 1+\gamma\lambda e_Q(x,a,t-1) & \text{if } (x_t,a_t) = (x,a) \end{cases} \tag{35}$$

As with e, all $e_Q(x,a,t)$'s must be reset to zero whenever a new trial is begun from a random starting state.

If π is a stochastic policy then it is better to replace (34) by

$$\Delta_Q(x,a) = r(x,a) + \gamma\tilde{V}(x_1) - \hat{Q}(x,a;v) \tag{36}$$

where

$$\tilde{V}(x_1) = \sum_{b\in A(x_1)} \text{Prob}\{\pi(x)=b\}\hat{Q}(x_1,b;v) \tag{37}$$

Rummery and Niranjan (1994) suggest the use of (34) even if π is stochastic; in that case, the $\pi(x_1)$ in (34) corresponds to an instance generated by the stochastic policy at x_1. We feel that, as an estimate of $V^\pi(x_1)$, $\tilde{V}(x_1)$ is better than the term $\hat{Q}(x_1,\pi(x_1);v)$ used in (34), and so it fits-in better with the definition of Q^π in (14). Also, if the the size of $A(x_1)$ is small then the computations of $\tilde{V}(x_1)$ is not much more expensive than that of $\hat{Q}(x_1,\pi(x_1);v)$.

5 Delayed Reinforcement Learning Methods

Dynamic Programming (DP) methods (Ross 1983; Bertsekas 1989) are well known classical tools for solving the stochastic optimal control problem formulated in §3. Since delayed RL methods also solve the same problem, how do they differ from DP methods?[8] Following are the main differences.

- Whereas DP methods simply aim to obtain the optimal value function and an optimal policy using off–line iterative methods, delayed RL methods aim to *learn the same concurrently with real time system operation* and improve performance over time.

[8]The connection between DP and delayed RL was first established by Werbos (1987, 1989, 1992) and Watkins (1989).

- DP methods deal with the complete state space, X in their computations, while delayed RL methods operate on $\tilde{X}$, the set of states that occur during real time system operation. In many applications X is very large, but $\tilde{X}$ is only a small, manageable subset of X. Therefore, in such applications, DP methods suffer from the *curse of dimensionality*, but delayed RL methods do not have this problem. Also, typically delayed RL methods employ function approximators (for value function, policy etc.) that generalize well, and so, after learning, they provide near optimal performance even on unseen parts of the state space.
- DP methods fundamentally require a system model. On the other hand, the main delayed RL methods are model-free; hence they are particularly suited for the on-line learning control of complicated systems for which a model is difficult to derive.
- Because delayed RL methods continuously learn in time they are better suited than DP methods for adapting to situations in which the system and goals are non-stationary.

Although we have said that delayed RL methods enjoy certain key advantages, we should also add that DP has been the fore-runner from which delayed RL methods obtained clues. In fact, it is correct to say that delayed RL methods are basically rearrangements of the computational steps of DP methods so that they can be applied during real time system operation.

Delayed RL methods can be grouped into two categories: model-based methods and model–free methods. Model based methods have direct links with DP. Model–free methods can be viewed as appropriate modifications of the model based methods so as to avoid the model requirement. These methods will be described in detail below.

5.1 Model Based Methods

In this subsection we discuss DP methods and their possible modification to yield delayed RL methods. There are two popular DP methods: value iteration and policy iteration. Value iteration easily extends to give a delayed RL method called 'real time DP'. Policy iteration, though it does not directly yield a delayed method, it forms the basis of an important model–free delayed RL method called actor–critic.

5.1.1 Value Iteration

The basic idea in value iteration is to compute $V^\star(x)$ as

$$V^\star(x) = \lim_{n\to\infty} V_n^\star(x) \tag{38}$$

where $V_n^\star(x)$ is the optimal value function over a finite-horizon of length n, i.e., $V_n^\star(x)$ is the maximum expected return if the decision task is terminated n steps after starting in state x. For $n = 1$, the maximum expected return is just the maximum of the expected immediate payoff:

$$V_1^\star(x) = \max_{a\in A(x)} r(x,a) \;\; \forall\, x \tag{39}$$

Then, the recursion,[9]

$$V^\star_{n+1}(x) = \max_{a \in A(x)} \left[r(x,a) + \gamma \sum_y P_{xy}(a) V^\star_n(y) \right] \quad \forall\, x \tag{40}$$

can be used to compute $V^\star_{n+1}$ for $n = 1, 2, \cdots$. (Iterations can be terminated after a large number (N) of iterations, and $V^\star_N$ can be taken to be a good approximation of $V^\star$.)

In value iteration, a policy is not involved. But it is easy to attach a suitable policy with a value function as follows. Associated with each value function, $V : X \rightarrow R$ is a policy, π that is *greedy with respect to* V, i.e.,

$$\pi(x) = \arg \max_{a \in A(x)} \left[r(x,a) + \gamma \sum_y P_{xy}(a) V(y) \right] \quad \forall\, x \tag{41}$$

If the state space, X has a very large size (e.g., size$=k^d$, where $d=$ number of components of x, k=number of values that each component can take, $d \approx 10$, $k \approx 100$) then value iteration is prohibitively expensive. This difficulty is usually referred to as the *curse of dimensionality.*

In the above, we have assumed that (38) is correct. Let us now prove this convergence. It turns out that convergence can be established for a more general algorithm, of which value iteration is a special case. We call this algorithm as *generalized value iteration.*

Generalized Value Iteration

Set $n = 1$ *and* $V^\star_1$ = *an arbitrary function over states.*
Repeat

1. *Choose a subset of states,* B_n *and set*

$$V^\star_{n+1}(x) = \begin{cases} \max_{a \in A(x)} \left[r(x,a) + \gamma \sum_y P_{xy}(a) V^\star_n(y) \right] & \text{if} \quad x \in B_n \\ V^\star_n(x) & \text{otherwise} \end{cases} \tag{42}$$

2. *Reset* $n := n + 1$.

If we choose $V^\star_1$ as in (39) and take $B_n = X$ for all n, then the above algorithm reduces to value iteration. Later we will go into other useful cases of generalized value iteration. But first, let us concern ourselves with the issue of convergence. If $x \in B_n$, we will say that the value of state x has been backed up at the n-th iteration. Proof of convergence is based on the following result (Bertsekas & Tsitsiklis 1989; Watkins 1989; Barto *et al* 1992).

Local Value Improvement Theorem

Let $M_n = \max_x |V^\star_n(x) - V^\star(x)|$. Then $\max_{x \in B_n} |V^\star_{n+1}(x) - V^\star(x)| \leq \gamma M_n$.

Proof: Take any $x \in B_n$. Let $a^\star = \pi^\star(x)$ and $a^\star_n = \pi^\star_n(x)$, where $\pi^\star_n$ is a policy that is greedy with respect to $V^\star_n$. Then

$$\begin{aligned} V^\star_{n+1}(x) &\geq r(x,a^\star) + \gamma \textstyle\sum_y P_{xy}(a^\star) V^\star_n(y) \\ &\geq r(x,a^\star) + \gamma \textstyle\sum_y P_{xy}(a^\star) \left[V^\star(y) - M \right] \\ &= V^\star(x) - \gamma M_n \end{aligned}$$

[9] One can also view the recursion as doing a fixed-point iteration to solve Bellman's optimality equation, (13).

Similarly,

$$\begin{aligned} V^\star_{n+1}(x) &= r(x,a^\star_n) + \gamma \textstyle\sum_y P_{xy}(a^\star_n) V^\star_n(y) \\ &\le r(x,a^\star_n) + \gamma \textstyle\sum_y P_{xy}(a^\star_n) \left[V^\star(y) + M\right] \\ &= V^\star(x) + \gamma M_n \end{aligned}$$

and so the theorem is proved. □

The theorem implies that $M_{n+1} \le M_n$ where $M_{n+1} = \max_x |V^\star_{n+1}(x) - V^\star(x)|$. A little further thought shows that the following is also true. If, at the end of iteration k, K further iterations are done in such a way that the value of each state is backed up at least once in these K iterations, i.e., $\cup_{n=k+1}^{k+K} B_n = X$, then we get $M_{k+K} \le \gamma M_k$. Therefore, *if the value of each state is backed up infinitely often, then (38) holds.*[10] In the case of value iteration, the value of each state is backed up at each iteration and so (38) holds.

Generalized value iteration was proposed by Bertsekas (1982, 1989) and developed by Bertsekas and Tsitsiklis (1989) as a suitable method of solving stochastic optimal control problems on multi-processor systems with communication time delays and without a common clock. If N processors are available, the state space can be partitioned into N sets – one for each processor. The times at which each processor backs up the values of its states can be different for each processor. To back up the values of its states, a processor uses the "present" values of other states communicated to it by other processors.

Barto, Bradtke and Singh (1992) suggested the use of generalized value iteration as a way of learning during real time system operation. They called their algorithm as *Real Time Dynamic Programming* (RTDP). In generalized value iteration as specialized to RTDP, n denotes system time. At time step n, let us say that the system resides in state x_n. Since $V^\star_n$ is available, a_n is chosen to be an action that is greedy with respect to $V^\star_n$, i.e., $a_n = \pi^\star_n(x_n)$. B_n, the set of states whose values are backed up, is chosen to include x_n and, perhaps some more states. In order to improve performance in the immediate future, one can do a lookahead search to some fixed search depth (either exhaustively or by following policy, $\pi^\star_n$) and include these probable future states in B_n. Because the value of x_n is going to undergo change at the present time step, it is a good idea to also include, in B_n, the most likely predecessors of x_n (Moore & Atkeson 1993).

One may ask: since a model of the system is available, why not simply do value iteration or, do generalized value iteration as Bertsekas and Tsitsiklis suggest? In other words, what is the motivation behind RTDP? The answer is simple. In most problems (e.g., playing games such as checkers and backgammon) the state space is extremely large, but only a small subset of it actually occurs during usage. Because RTDP works concurrently with actual system operation, it focusses on regions of the state space that are most relevant to the system's behaviour. For instance, successful learning was accomplished in the checkers program of Samuel (1959) and in the backgammon program, TDgammon of Tesauro (1992) using variations of RTDP. In (Barto *et al* 1992), Barto, Bradtke and Singh also use RTDP to make interesting connections and useful extensions to learning real time search algorithms in Artificial Intelligence (Korf 1990).

The convergence result mentioned earlier says that the values of all states have to

[10] If $\gamma = 1$, then convergence holds under certain assumptions. The analysis required is more sophisticated. See (Bertsekas & Tsitsiklis 1989; Bradtke 1994) for details.

be backed up infinitely often[11] in order to ensure convergence. So it is important to suitably explore the state space in order to improve performance. Barto, Bradtke and Singh have suggested two ways of doing exploration[12]: (1) adding stochasticity to the policy; and (2) doing learning cumulatively over multiple trials.

If, only an inaccurate system model is available then it can be updated in real time using a system identification technique, such as maximum likelihood estimation method (Barto *et al* 1992). The current system model can be used to perform the computations in (42). Convergence of such adaptive methods has been proved by Gullapalli and Barto (1994).

5.1.2 Policy Iteration

Policy iteration operates by maintaining a representation of a policy and its value function, and forming an improved policy using them. Suppose π is a given policy and V^π is known. How can we improve π? An answer will become obvious if we first answer the following simpler question. If μ is another given policy then when is

$$V^\mu(x) \geq V^\pi(x) \;\; \forall\, x \tag{43}$$

i.e., when is μ uniformly better than π? The following simple theorem (Watkins 1989) gives the answer.

Policy Improvement Theorem
The policy μ is uniformly better than policy π if

$$Q^\pi(x, \mu(x)) \geq V^\pi(x) \;\; \forall\, x \tag{44}$$

Proof: To avoid clumsy details let us give a not-so-rigorous proof (Watkins 1989). Starting at x, it is better to follow μ for one step and then to follow π, than it is to follow π right from the begining. By the same argument, it is better to follow μ for one further step from the state just reached. Repeating the argument we get that it is always better to follow μ than π. See Bellman and Dreyfus (1962) and Ross (1983) for a detailed proof. $\square$

Let us now return to our original question: given a policy π and its value function V^π, how do we form an improved policy, μ? If we define μ by

$$\mu(x) = \arg \max_{a \in A(x)} Q^\pi(x, a) \;\; \forall\, x \tag{45}$$

then (44) holds. By the policy improvement theorem μ is uniformly better than π. This is the main idea behind policy iteration.

Policy Iteration
Set π := an arbitrary initial policy and compute V^π.
Repeat

1. *Compute Q^π using (14).*
2. *Find μ using (45) and compute V^μ.*

---

[11] For good practical performance it is sufficient that states that are most relevant to the system's behaviour are backed up repeatedly.

[12] Thrun (1986) has discussed the importance of exploration and suggested a variety of methods for it

3. Set: $\pi := \mu$ and $V^\pi := V^\mu$.

until $V^\mu = V^\pi$ occurs at step 2.

Nice features of the above algorithm are: (1) it terminates after a finite number of iterations because there are only a finite number of policies; and (2) when termination occurs we get

$$V^\pi(x) = \max_a Q^\pi(x,a) \;\; \forall x$$

(i.e., V^π satisfies Bellman's optimality equation) and so π is an optimal policy. But the algorithm suffers from a serious drawback: it is very expensive because the entire value function associated with a policy has to be recalculated at each iteration (step 2). Even though V^μ may be close to V^π, unfortunately there is no simple short cut to compute it. In §5.2 we will discuss a well-known model–free method called the *actor-critic* method which gives an inexpensive approximate way of implementing policy iteration.

5.2 Model–Free Methods

Model–free delayed RL methods are derived by making suitable approximations to the computations in value iteration and policy iteration, so as to eliminate the need for a system model. Two important methods result from such approximations: Barto, Sutton and Anderson's actor–critic (Barto *et al* 1983), and Watkins' Q–Learning (Watkins 1989). These methods are milestone contributions to the optimal feedback control of dynamic systems.

5.2.1 Actor-Critic Method

The actor-critic method was proposed by Barto, Sutton and Anderson (1983) (in their popular work on balancing a pole on a moving cart) as a way of combining, on a step-by-step basis, the process of forming the value function with the process of forming a new policy. The method can also be viewed as a practical, approximate way of doing policy iteration: perform one step of an on-line procedure for estimating the value function for a given policy, and at the same time perform one step of an on-line procedure for improving that policy. The actor-critic method[13] is best derived by combining the ideas of §2 and §4 on immediate RL and estimating value function, respectively. Details are as follows.

Actor (π) Let m denote the total number of actions. Maintain an approximator, $g(\cdot; w) : X \rightarrow R^m$ so that $z = g(x; w)$ is a vector of merits of the various feasible actions at state x. In order to do exploration, choose actions according to a stochastic action selector such as (6).[14]

Critic (V^π) Maintain an approximator, $\hat{V}(\cdot; w) : X \rightarrow R$ that estimates the value function (expected total reward) corresponding to the stochastic policy mentioned above. The ideas of §4 can be used to update $\hat{V}$.

Let us now consider the process of learning the actor. Unlike immediate RL, learning is more complicated here for the following reason. Whereas, in immediate

[13] A mathematical analysis of this method has been done by Williams and Baird (1990).

[14] In their original work on pole-balancing, Barto, Sutton and Anderson suggested a different way of including stochasticity.

RL the environment immediately provides an evaluation of an action, in delayed RL the effect of an action on the total reward is not immediately available and has to be estimated appropriately. Suppose, at some time step, the system is in state x and the action selector chooses action a^k. For g the learning rule that parallels (5) would be

$$g_k(x; w) := g_k(x; w) + \alpha \left[\rho(x, a^k) - \hat{V}(x; v)\right] \tag{46}$$

where $\rho(x; a^k)$ is the expected total reward obtained if a^k is applied to the system at state x and then policy π is followed from the next step onwards. An approximation is

$$\rho(x, a^k) \approx r(x, a^k) + \gamma \sum_y P_{xy}(a^k) \hat{V}(y; v) \tag{47}$$

This estimate is unavailable because we do not have a model. A further approximation is

$$\rho(x, a^k) \approx r(x, a^k) + \gamma \hat{V}(x_1; v) \tag{48}$$

where x_1 is the state occuring in the real time operation when action a^k is applied at state x. Using (48) in (46) gives

$$g_k(x; w) := g_k(x; w) + \alpha \Delta(x) \tag{49}$$

where Δ is as defined in (29). The following algorithm results.

Actor–Critic Trial

Set $t = 0$ and $x =$a random starting state.
Repeat (for a number of time steps)

1. *With the system at state, x, choose action a according to (6) and apply it to the system. Let x_1 be the resulting next state.*
2. *Compute $\Delta(x) = r(x, a) + \gamma \hat{V}(x_1; v) - \hat{V}(x; v)$*
3. *Update $\hat{V}$ using $\hat{V}(x; v) := \hat{V}(x; v) + \beta \Delta(x)$*
4. *Update g_k using (49) where k is such that $a = a^k$.*

The above algorithm uses the $TD(0)$ estimate of V^π. To speed–up learning the $TD(\lambda)$ rule, (31) can be employed. Barto, Sutton and Anderson (1983) and others (Gullapalli 1992a; Gullapalli *et al* 1994) use the idea of eligibility traces for updating g also. They give only an intuitive explanation for this usage. Lin (1992) has suggested the accumulation of data until a trial is over, update $\hat{V}$ using (28) for all states visited in the trial, and then update g using (49) for all (state,action) pairs experienced in the trial.

5.2.2 Q–Learning

Just as the actor–critic method is a model-free, on-line way of approximately implementing policy iteration, Watkins' Q–Learning algorithm is a model-free, on-line way of approximately implementing generalized value iteration. Though the RTDP algorithm does generalized value iteration concurrently with real time system operation, it requires the system model for doing a crucial operation: the determination of the maximum on the right hand side of (42). Q–Learning overcomes this problem elegantly by operating with the Q–function instead of the value function. (Recall, from §3, the definition of Q–function and the comment on its advantage over value function.)

The aim of Q–Learning is to find a function approximator, $\hat{Q}(\cdot,\cdot;v)$ that approximates $Q^\star$, the solution of Bellman's optimality equation, (16), in on-line mode without employing a model. However, for the sake of developing ideas systematically, let us begin by assuming that a system model is available and consider the modification of the ideas of §5.1.1 to use the Q–function instead of the value function. If we think in terms of a function approximator, $\hat{V}(x;v)$ for the value function, the basic update rule that is used throughout §5.1.1 is

$$\hat{V}(x;v) := \max_{a\in A(x)} \left[r(x,a) + \gamma \sum_y P_{xy}(a)\hat{V}(y;v) \right]$$

For the Q–function, the corresponding rule is

$$\hat{Q}(x,a;v) := r(x,a) + \gamma \sum_y P_{xy}(a) \max_{b\in A(y)} \hat{Q}(y,b;v) \tag{50}$$

Using this rule, all the ideas of §5.1.1 can be easily modified to employ the Q–function.

However, our main concern is to derive an algorithm that avoids the use of a system model. A model can be avoided if we: (1) replace the summation term in (50) by $\max_{b\in A(x_1)} \hat{Q}(x_1,b;v)$ where x_1 is an instance of the state resulting from the application of action a at state x; and (2) achieve the effect of the update rule in (50) via the "averaging" learning rule,

$$\hat{Q}(x,a;v) := \hat{Q}(x,a;v) + \beta \left[r(x,a) + \gamma \max_{b\in A(x_1)} \hat{Q}(x_1,b;v) - \hat{Q}(x,a;v) \right] \tag{51}$$

If (51) is carried out we say that the Q–value of (x,a) has been backed up. Using (51) in on-line mode of system operation we obtain the Q–Learning algorithm.

Q–Learning Trial
Set $t=0$ and $x=$ a random starting state.
Repeat (for a number of time steps)

1. *Choose action $a\in A(x)$ and apply it to the system. Let x_1 be the resulting state.*
2. *Update $\hat{Q}$ using (51).*
3. *Reset $x := y$.*

The remark made below equation, (7) in §2 is very appropriate for the learning rule, (51). Watkins showed[15] that *if the Q–value of each admissible (x,a) pair is backed up infinitely often, and if the step size, β is decreased to zero in a suitable way then as $t\to\infty$, $\hat{Q}$ converges to $Q^\star$ with probability one.* Practically, learning can be achieved by: (1) using, in step 1, an appropriate exploration policy that tries all

[15] A revised proof was given by Watkins and Dayan (1992). Tsitsiklis (1993) and Jaakkola *et al* , (1994) have given other proofs.

actions;[16] (2) doing multiple trials to ensure that all states are frequently visited; and (3) decreasing β towards zero as learning progresses.

We now discuss a way of speeding up Q–Learning by using the $TD(\lambda)$ estimate of the Q–function, derived in §4. If $TD(\lambda)$ is to be employed in a Q–Learning trial, a fundamental requirement is that the policy used in step 1 of the Q–Learning Trial and the policy used in the update rule, (51) should match (note the use of π in (34) and (37)). Thus $TD(\lambda)$ can be used if we employ the greedy policy,

$$\pi(x) = \arg \max_{a \in A(x)} \hat{Q}(x, a; v) \tag{52}$$

in step 1.[17] But, this leads to a problem: use of the greedy policy will not allow exploration of the action space, and hence poor learning can occur. Rummery and Niranjan (1994) give a nice comparitive account of various attempts described in the literature for dealing with this conflict. Here we only give the details of an approach that Rummery and Niranjan found to be very promising.

Consider the stochastic policy (based on the Boltzmann distribution and Q-values),

$$\text{Prob}\{\pi(x) = a | x\} = \frac{exp(\hat{Q}(x, a; v)/T)}{\sum_{b \in A(x)} exp(\hat{Q}(x, b; v)/T)} \,, \quad a \in A(x) \tag{53}$$

where $T \in [0, \infty)$. When $T \rightarrow \infty$ all actions have equal probabilities and, when $T \rightarrow 0$ the stochastic policy tends towards the greedy policy in (52). To learn, T is started with a suitable large value (depending on the initial size of the Q–values) and is decreased to zero using an annealing rate; at each T thus generated, multiple Q–learning trials are performed. This way, exploration takes place at the initial large T values. The $TD(\lambda)$ learning rule, (36) estimates expected returns for the policy at each T, and, as $T \rightarrow 0$, $\hat{Q}$ will converge to $Q^\star$. The ideas here are somewhat similar to those of simulated annealing.

5.3 Extension To Continuous Spaces

Optimal control of dynamic systems typically involves the solution of delayed RL problems having continuous state/action spaces. If the state space is continuous but the action space is discrete then all the delayed RL algorithms discussed earlier can be easily extended, provided appropriate function approximators that generalize a real time experience at a state to all topologically nearby states are used; see §6 for a discussion of such approximators. On the other hand, if the action space is continuous, extension of the algorithms is more difficult. The main cause of the difficulty can be easily seen if we try extending RTDP to continuous action spaces: the max operation in (42) is non–trivial and difficult if $A(x)$ is continuous. (Therefore, even methods based on value iteration need to maintain a function approximator for actions.) In the rest of this subsection we will give a brief review of

[16] Note that step 1 does not put any restriction on choosing a feasible action. So, any stochastic exploration policy that, at every x generates each feasible action with positive probability can be used. When learning is complete, the greedy policy, $\pi(x) = \arg\max_{a \in A(x)} \hat{Q}(x, a; v)$ should be used for optimal system performance.

[17] If more than one action attains the maximum in (52) then for convenience we take π to be a stochastic policy that makes all such maximizing actions equally probable.

various methods of handling continuous action spaces. Just to make the presentation easy, we will make the following assumptions.

- The system being controlled is deterministic. Let

$$x_{t+1} = f(x_t, a_t) \tag{54}$$

describe the transition.[18]

- There are no action constraints, i.e., $A(x)$ =an m–dimensional real space for every x.

- All functions involved (r, f, $\hat{V}$, $\hat{Q}$ etc.) are continuously differentiable.

Let us first consider model-based methods. Werbos (1990b) has proposed a variety of algorithms. Here we will describe only one important algorithm, the one that Werbos refers to as *Backpropagated Adaptive Critic*. The algorithm is of the actor–critic type, but it is somewhat different from the actor–critic method of §5.2.1. There are two function approximators: $\hat{\pi}(\cdot; w)$ for action; and, $\hat{V}(\cdot; v)$ for critic. The critic is meant to approximate $V^{\hat{\pi}}$; at each time step, it is updated using the $TD(\lambda)$ learning rule, (31) of §4. The actor tries to improve the policy at each time step using the hint provided by the policy improvement theorem in (44). To be more specific, let us define

$$Q(x, a) \stackrel{\text{def}}{=} r(x, a) + \gamma \hat{V}(f(x, a); v) \tag{55}$$

At time t, when the system is at state x_t, we choose the action, $a_t = \hat{\pi}(x_t; w)$, leading to the next state, x_{t+1} given by (54). Let us assume $\hat{V} = V^{\hat{\pi}}$, so that $V^{\hat{\pi}}(x_t) = Q(x_t, a_t)$ holds. Using the hint from (44), we aim to adjust $\hat{\pi}(x_t; w)$ to give a new value, a^{new} such that

$$Q(x_t, a^{\text{new}}) > Q(x_t, a_t) \tag{56}$$

A simple learning rule that achieves this requirement is

$$\hat{\pi}(x_t; w) := \hat{\pi}(x_t; w) + \alpha \frac{\partial Q(x_t, a)}{\partial a}|_{a=a_t} \tag{57}$$

where α is a small (positive) step size. The partial derivative in (57) can be evaluated using

$$\frac{\partial Q(x_t, a)}{\partial a} = \frac{\partial r(x_t, a)}{\partial a} \quad + \quad \gamma \frac{\partial \hat{V}(y; v)}{\partial y}|_{y=f(x_t,a)} \frac{\partial f(x_t, a)}{\partial a} \tag{58}$$

Let us now come to model–free methods. A simple idea is to adapt a function approximator, $\hat{f}$ for the system model function, f, and use $\hat{f}$ instead of f in Werbos' algorithm. On–line experience, i.e., the combination, (x_t, a_t, x_{t+1}), can be used to learn $\hat{f}$. This method was proposed by Brody (1992), actually as a way of overcoming a serious deficiency[19] associated with an ill–formed model–free method suggested by Jordan and Jacobs (1990). A key difficulty associated with Brody's method is that, until the learning system adapts a good $\hat{f}$, system performance does not improve

[18] Werbos (1990b) describes ways of treating stochastic systems.
[19] This deficiency was also pointed out by Gullapalli (1992b).

at all; in fact, at the early stages of learning the method can perform in a confused way. To overcome this problem Brody suggests that $\hat{f}$ be learnt well, before it is used to train the actor and the critic.

A more direct model–free method can be derived using the ideas of §5.2.1 and employing a learning rule similar to (8) for adapting $\hat{\pi}$. This method was proposed and successfully demonstrated by Gullapalli (Gullapalli 1992a; Gullapalli *et al* 1994). Since Gullapalli's method learns by observing the effect of a randomly chosen perturbation of the policy, it is not as systematic as the policy change in Brody's method.

We now propose a new model–free method that systematically changes the policy similar to what Brody's method does, and, avoids the need for adapting a system model. This is achieved using a function approximator, $\hat{Q}(\cdot, \cdot; v)$ for approximating $Q^{\hat{\pi}}$, the Q–function associated with the actor. The $TD(\lambda)$ learning rule in (33) can be used for updating $\hat{Q}$. Also, policy improvement can be attempted using the learning rule (similar to (57)),

$$\hat{\pi}(x_t; w) := \hat{\pi}(x_t; w) + \alpha \frac{\partial \hat{Q}(x_t, a)}{\partial a}|_{a=a_t} \tag{59}$$

We are currently performing simulations to study the performance of this new method relative to the other two model–free methods mentioned above.

Werbos' algorithm and our Q-Learning based algorithm are deterministic, while Gullapalli's algorithm is stochastic. The deterministic methods are expected to be much faster, whereas the stochastic method has better assurance of convergence to the true solution. The arguments are similar to those mentioned at the end of §2.

6 Other Issues

In this section we discuss a number of issues relevant to practical implementation of RL algorithms. A nice discussion of these (and other) issues has been presented by Barto (1992).

6.1 Function–Approximation

A variety of function approximators has been employed by researchers to practically solve RL problems. When the input space of the function approximator is finite, the most straight–forward method is to use a *look–up table* (Singh 1992a; Moore & Atkeson 1993). All theoretical results on the convergence of RL algorithms assume this representation. The disadvantage of using look–up table is that if the input space is large then the memory requirement becomes prohibitive.[20] Continuous input spaces have to be discretized when using a look–up table. If the discretization is done finely so as to obtain good accuracy we have to face the 'curse of dimensionality'. One way of overcoming this is to do a problem–dependent discretization; see, for example, the 'BOXES' representation used by Barto, Sutton and Anderson (1983) and others (Michie & Chambers 1968; Gullapalli *et al* 1994; Rosen *et al* 1991) to solve the pole balancing problem.

[20] Buckland and Lawrence (1994) have proposed a new delayed RL method called Transition point DP which can significantly reduce the memory requirement for problems in which optimal actions change infrequently in time.

Non look–up table approaches use parametric function approximation methods. These methods have the advantage of being able to generalize beyond the training data and hence give reasonable performance on unvisited parts of the input space. Among these, connectionist methods are the most popular. Connectionist methods that have been employed for RL can be classified into four groups: multi–layer perceptrons; methods based on clustering; CMAC; and recurrent networks. *Multi–layer perceptrons* have been successfully used by Anderson (1986, 1989) for pole balancing, Lin (1991a, 1991b, 1991c, 1992) for a complex test problem, Tesauro (1992) for backgammon, Thrun (1993) and Millan and Torras (1992) for robot navigation, and others (Boyen 1992; Gullapalli *et al* 1994). On the other hand, Watkins (1989), Chapman (1991), Kaelbling (1990, 1991), and Shepanski and Macy (1987) have reported bad results. A modified form of Platt's *Resource Allocation Network* (Platt 1991), a method based on radial basis functions, has been used by Anderson (1993) for pole balancing. Many researchers have used *CMAC* (Albus 1975) for solving RL problems: Watkins (1989) for a test problem; Singh (1991, 1992b, 1992d) and Tham and Prager (1994) for a navigation problem; Lin and Kim (1991) for pole balancing; and Sutton (1990, 1991b) in his 'Dyna' architecture. Recurrent networks with context information feedback have been used by Bacharach (1991, 1992) and Mozer and Bacharach (1990a, 1990b) in dealing with RL problems with incomplete state information.

A few non–connectionist methods have also been used for RL. Mahadevan and Connell (1991) have used statistical clustering in association with Q–Learning for the automatic programming of a mobile robot. A novel feature of their approach is that the number of clusters is dynamically varied. Chapman and Kaelbling (1991) have used a tree–based clustering approach in combination with a modified Q–Learning algorithm for a difficult test problem with a huge input space.

The function approximator has to exercise care to ensure that learning at some input point, x does not seriously disturb the function values for $y \neq x$. It is often advantageous to choose a function approximator and employ an update rule in such a way that the function values of x and states 'near' x are modified similarly while the values of states 'far' from x are left unchanged. Such a choice usually leads to good generalization, i.e., good performance of the learnt function approximator even on states that are not visited during learning. In this respect, CMAC and methods based on clustering, such as RBF, statistical clustering, etc., are more suitable than multi–layer perceptrons.

The effect of errors introduced by function approximators on the optimal performance of the controller has not been well understood.[21] It has been pointed out by Watkins (1989), Bradtke (1993), and others (Barto 1992), that, if function approximation is not done in a careful way, poor learning can result. In the context of Q–Learning, Thrun and Schwartz (1993) have shown that errors in function approximation can lead to a systematic over estimation of the Q–function. Linden (1993) points out that in many problems the value function is discontinuous and so using continuous function approximators is inappropriate. But he does not suggest any clear remedies for this problem. Overall, it must be mentioned that much work needs to be done on the use of function approximators for RL.

[21] Bertsekas(1989) and Singh and Yee (1993) have derived some theoretical bounds for errors in value function in terms of function approximator error.

6.2 Modular and Hierarchical Architectures

When applied to problems with large task space or sparse rewards, RL methods are terribly slow to learn. Dividing the problem into simpler subproblems, using a hierarchical control structure, etc., are ways of overcoming this.

Sequential task decomposition is one such method. This method is useful when a number of complex tasks can be performed making use of a finite number of "elemental" tasks or skills, say, $T_1, T_2, \cdots, T_n$. The original objective of the controller can then be achieved by temporally concatenating a number of these elemental tasks to form what is called a "composite" task. For example,

$$C_j = [T(j,1), T(j,2), \cdots, T(j,k)]\,, \quad \text{where} \quad T(j,i) \in \{T_1, T_2, \cdots, T_n\}$$

is a composite task made up of k elemental tasks that have to be performed in the order listed. Reward functions are defined for each of the elemental tasks, making them more abundant than in the original problem definition.

Singh (1992a, 1992b) has proposed an algorithm based on a modular connectionist network (Jacobs *et al* 1991), making use of these ideas. In his work the controller is unaware of the decomposition of the task and has to learn both the elemental tasks, and the decomposition of the composite tasks simultaneously. Tham and Prager (1994) and Lin (1993) have proposed similar solutions. Mahadevan and Connell (1991) have developed a method based on the *subsumption architecture* (Brooks 1986) where the decomposition of the task is specified by the user before hand, and the controller learns only the elemental tasks, while Maes and Brooks (1990) have shown that the controller can be made to learn the decomposition also, in a similar framework. All these methods require some external agency to specify the problem decomposition. Can the controller itself learn how the problem is to be decomposed? Though Singh (1992d) has some preliminary results, much work needs to be done here.

Another approach to this problem is to use some form of hierarchical control (Watkins 1989). Here there are different "levels" of controllers[22], each learning to perform a more abstract task than the level below it and directing the lower level controllers to achieve its objective. For example, in a ship a navigator decides in what direction to sail so as to reach the port while the helmsman steers the ship in the direction indicated by the navigator. Here the navigator is the higher level controller and the helmsman the lower level controller. Since the higher level controllers have to work on a smaller task space and the lower level controllers are set simpler tasks improved performance results.

Examples of such hierarchical architectures are *Feudal RL* by Dayan and Hinton (1993) and *Hierarchical planning* by Singh (1992a, 1992c). These methods too, require an external agency to specify the hierarchy to be used. This is done usually by making use of some "structure" in the problem.

Training controllers on simpler tasks first and then training them to perform progressively more complex tasks using these simpler tasks, can also lead to better performance. Here at any one stage the controller is faced with only a simple learning task. This technique is called *shaping* in animal behaviour literature. Gullapalli (1992a) and Singh (1992d) have reported some success in using this idea. Singh shows that the controller can be made to "discover" a decomposition of the task by itself using this technique.

[22] Controllers at different levels may operate at different temporal resolutions.

6.3 Speeding–Up Learning

Apart from the ideas mentioned above, various other techniques have been suggested for speeding–up RL. Two novel ideas have been suggested by Lin (1991a, 1991b, 1991c, 1992): *experience playback*; and *teaching*. Let us first discuss experience playback. An experience consists of a quadruple (occuring in real time system operation), (x, a, y, r), where x is a state, a is the action applied at state x, y is the resulting state, and r is $r(x, a)$. Past experiences are stored in a finite memory buffer, $\mathcal{P}$. An appropriate strategy can be used to maintain $\mathcal{P}$. At some point in time let π be the "current" (stochastic) policy. Let

$$\mathcal{E} = \{(x, a, y, r) \in \mathcal{P} \mid \text{Prob}\{\pi(x) = a\} \geq \epsilon\}$$

where ϵ is some chosen tolerance. The learning update rule is applied, not only to the current experience, but also to a chosen subset of $\mathcal{E}$. Experience playback can be especially useful in learning about rare experiences. In teaching, the user provides the learning system with experiences so as to expedite learning.

Incorporating domain specific knowledge also helps in speeding–up learning. For example, for a given problem, a "nominal" controller that gives reasonable performance may be easily available. In that case RL methods can begin with this controller and improve its performance (Singh *et al* 1994). Domain specific information can also greatly help in choosing state representation and setting up the function approximators (Barto 1992; Millan & Torras 1992).

In many applications an inaccurate system model is available. It turns out to be very inefficient to discard the model and simply employ a model–free method. An efficient approach is to interweave a number of "planning" steps between every two on-line learning steps. A planning step may be one of the following: a time step of a model–based method such as RTDP; or, a time step of a model–free method for which experience is generated using the available system model. In such an approach, it is also appropriate to adapt the system model using on–line experience. These ideas form the basis of Sutton's *Dyna* architectures (Sutton 1990, 1991b) and related methods (Moore & Atkeson 1993; Peng & Williams 1993).

7 Conclusion

In this paper we have tried to give a cohesive overview of existing RL algorithms. Though research has reached a mature level, RL has been successfully demonstrated only on a few practical applications (Gullapalli *et al* 1994; Tesauro 1992; Mahadevan & Connell 1991; Thrun 1993), and clear guidelines for its general applicability do not exist. The connection between DP and RL has nicely bridged control theorists and AI researchers. With contributions from both these groups on the pipeline, more interesting results are forthcoming and it is expected that RL will make a strong impact on the intelligent control of dynamic systems.

References

J .S. Albus, 1975, A new approach to manipulator control: The cerebellar model articulation controller (CMAC). *Trans. ASME, J. Dynamic Sys., Meas., Contr.*, 97:220–227.

C. W. Anderson, 1986, *Learning and Problem Solving with Mulitlayer Connectionist Systems.* Ph.D. thesis, University of Massachusetts, Amherst, MA, USA.

C. W. Anderson, 1987, Strategy learning with multilayer connectionist representations. Technical report, TR87–509.3, GTE Laboratories, INC., Waltham, MA, USA.

C. W. Anderson, April 1989, Learning to control an inverted pendulum using neural networks. *IEEE Control Systems Magazine*, pages 31–37.

C. W. Anderson, 1993, Q–Learning with hidden–unit restarting. In *Advances in Neural Information Processing Systems 5*, S. J. Hanson, J. D. Cowan, and C. L. Giles, editors, pages 81–88, Morgan Kaufmann, San Mateo, CA, USA.

J. R. Bacharach, 1991, A connectionist learning control architecture for navigation. In *Advances in Neural Information Processing Systems 3*, R. P. Lippman, J. E. Moody, and D. S. Touretzky, editors, pp. 457–463, Morgan Kaufmann, San Mateo, CA, USA.

J. R. Bacharach, 1992, *Connectionist modeling and control of finite state environments.* Ph.D. Thesis, University of Massachusetts, Amherst, MA, USA.

A. G. Barto, 1985, Learning by statistical cooperation of self–interested neuron–like computing elements. *Human Neurobiology*, 4:229–256.

A. G. Barto, 1986, Game–theoritic cooperativity in networks of self interested units. In *Neural Networks for Computing*, J. S. Denker, editor, pages 41–46, American Institute of Physics, New York, USA.

A. G. Barto, 1992, Reinforcemnet learning and adaptive critic methods. In *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*, D. A. White and D. A. Sofge, editors, pages 469–491, Van Nostrand Reinhold, New York, USA.

A. G. Barto and P. Anandan, 1985, Pattern recognizing stocahstic learning automata. *IEEE Transactions on Systems, Man, and Cybernatics*, 15:360–375.

A. G. Barto, P. Anandan, and C. W. Anderson, 1985, Cooperativity in networks of pattern recognizing stochastic learning automata. In *Proceedings of the Fourth Yale Workshop on Applications of Adaptive Systems Theory*, New Haven, CT, USA.

A. G. Barto, S. J. Bradtke, and S. P. Singh, 1992, Real–time learning and control using asynchronous dynamic programming. Technical Report COINS 91–57, University of Massachusetts, Amherst, MA, USA.

A. G. Barto and M. I. Jordan, 1987, Gradient following without back–propagation in layered networks. In *Proceedings of the IEEE First Annual Conference on Neural Networks*, M. Caudill and C. Butler, editors, pages II629–II636, San Diego, CA, USA.

A. G. Barto and S. P. Singh, 1991, On the computational economics of reinforcement learning. In *Connectionist Models Proceedings of the 1990 Summer School*, D. S. Touretzky, J. L. Elman, T. J. Sejnowski, and G. E. Hinton, editors, pages 35–44, Morgan Kaufmann, San Mateo, CA, USA.

A. G. Barto and R. S. Sutton, 1981, Landmark learning: an illustration of associative search. *Biological Cybernetics*, 42:1–8.

A. G. Barto and R. S. Sutton, 1982, Simulation of anticipatory responses in classical conditioning by a neuron–like adaptive element. *Behavioural Brain Research*, 4:221–235.

A. G. Barto, R. S. Sutton, and C. W. Anderson, 1983, Neuronlike elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man and Cybernetics*, 13:835–846.

A. G. Barto, R. S. Sutton, and P. S. Brouwer, 1981, Associative search network: a reinforcement learning associative memory. *IEEE Transactions on Systems, Man, and Cybernetics*, 40:201–211.

A. G. Barto, R. S. Sutton, and C. J. C. H. Watkins, 1990, Learning and sequential decision making. In *Learning and Computational Neuroscience: Foundations of Adaptive Networks*, M. Gabriel and J. Moore, editors, pages 539–602, MIT Press, Cambridge, MA, USA.

R. E. Bellman and S. E. Dreyfus, 1962, *Applied Dynamic Programming.* RAND Corporation.

D. P. Bertsekas, 1982, Distributed Dynamic Programming. *IEEE Transactions on Automatic Control*, 27:610–616.

D. P. Bertsekas, 1989, *Dynamic Programming: Deterministic and Stochastic Models.* Prentice–Hall, Englewood Cliffs, NJ, USA.

D. P. Bertsekas and J. N. Tsitsiklis, 1989, *Parallel and Distributed Computation: Numerical Methods.* Prentice–Hall, Englewood Cliffs, NJ, USA.

J. Boyen, 1992, *Modular Neural Networks for Learning Context–dependent Game Strategies.* Masters Thesis, Computer Speech and Language Processing, University of Cambridge, Cambridge, England.

S. J. Bradtke, 1993, Reinforcement learning applied to linear quadratic regulation. In *Advances in Neural Information Processing Systems 5*, S. J. Hanson, J. D. Cowan and C. L. Giles, editors, pages 295–302, Morgan Kaufmann, San Mateo, CA, USA.

S. J. Bradtke, 1994, *Incremental Dynamic Programming for On–line Adaptive Optimal Control.* CMPSCI Technical Report 94–62.

C. Brody, 1992, Fast learning with predictive forward models. In *Advances in Neural Information Processing Systems 4*, J. E. Moody, S. J. Hanson, and R. P. Lippmann, editors, pages 563–570, Morgan Kaufmann, San Mateo, CA, USA.

R. A. Brooks, 1986, Achieving artificial intelligence through building robots. Technical Report, A.I. Memo 899, Massachusetts Institiute of Technology, Aritificial Intelligence Laboratory, Cambridge, MA, USA.

K. M. Buckland and P. D. Lawrence, 1994, Transition point dynamic programming. In *Advances in Neural Information Processing Systems 6*, J. D. Cowan, G. Tesauro, and J. Alspector, editors, pages 639–646, Morgan Kaufmann, San Fransisco, CA, USA.

D. Chapman, 1991, *Vision, Instruction, and Action.* MIT Press.

D. Chapman and L. P. Kaelbling, 1991, Input generalization in delayed reinforcement learning: an algorithm and performance comparisions. In *Proceedings of the 1991 International Joint Conference on Artificial Intelligence.*

L. Chrisman, 1992, Planning for closed–loop execution using partially observable markovian decision processes. In *Proceedings of AAAI.*

P. Dayan, 1991a, Navigating through temporal difference. In *Advances in Neural Information Processing Systems 3*, R. P. Lippmann, J. E. Moody, and D. S. Touretzky, editors, pages 464–470, Morgan Kaufmann, San Mateo, CA, USA.

P. Dayan, 1991b, *Reinforcing Connectionism: Learning the Statistical Way.* Ph.D. Thesis, University of Edinburgh, Edinburgh, Scotland.

P. Dayan and G. E. Hinton, 1993, Feudal reinforcement learning. In *Advances in Neural Information Processing Systems 5*, S. J. Hanson, J. D. Cowan, and C. L. Giles, editors, pages 271–278, Morgan Kaufmann, San Mateo, CA, USA.

P. Dayan and T. J. Sejnowski, 1993, TD(λ) converges with probability 1. Technical Report, CNL, The Salk Institute, San Diego, CA, USA.

T. L. Dean and M. P. Wellman, 1991, *Planning and Control.* Morgan Kaufmann, San Mateo, CA, USA.

V. Gullapalli, 1990, A stochastic reinforcement algorithm for learning real–valued functions. *Neural Networks*, 3:671–692.

V. Gullapalli, 1992a, Reinforcement learning and its application to control. Technical Report, COINS, 92–10, Ph. D. Thesis, University of Massachusetts, Amherst, MA, USA.

V. Gullapalli, 1992b, A comparison of supervised and reinforcement learning methods on a reinforcment learning task. In *Proceedings of the 1991 IEEE Symposium on Intelligent Control*, Arlignton, VA, USA.

V. Gullapalli and A. G. Barto, 1994, Convergence of indirect adaptive asynchronous value iteration algorithms. In *Advances in Neural Information Processing Systems 6*, J. D. Cowan, G. Tesauro, and J. Alspector, editors, pages 695–702, Morgan Kaufmann, San Fransisco, CA, USA.

V. Gullapalli, J. A. Franklin, and H. Benbrahim, February 1994, Acquiring robot skills via reinforcement learning. *IEEE Control Systems Magazine*, pages 13–24.

J. A. Hertz, A. S. Krogh, and R. G. Palmer, 1991, *Introduction to the Theory of Neural Computation.* Addison–Wesley, CA, USA.

T. Jaakkola, M. I. Jordan, and S. P. Singh, 1994, Convergence of stochastic iterative dynamic programming algorithms. In *Advances in Neural Information Processing Systems 6*, J. D. Cowan, G. Tesauro, and J. Alspector, editors, pp. 703–710, Morgan Kaufmann, San Fransisco, CA, USA.

R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton, 1991, Adaptive mixtures of local experts. *Neural Computation*, 3:79–87.

M. I. Jordan and R. A. Jacobs, 1990, Learning to control an unstable system with forward modeling. In *Advances in Neural Information Processing Systems 2*, D. S. Touretzky, editor, Morgan Kaufmann, San Mateo, CA, USA.

M. I. Jordan and D. E. Rumelhart, 1990, Forward models: Supervised learning with a distal teacher. *Center for Cognitive Science Occasional Paper # 40*, Massachusetts Institute of Technology, Cambridge, MA, USA.

L. P. Kaelbling, 1990, Learning in embedded systems. Technical Report, TR–90–04, Ph.D. Thesis, Department of Computer Science, Stanford University, Stanford, CA, USA.

L. P. Kaelbling, 1991, *Learning in Embedded Systems.* MIT Press, Cambridge, MA, USA.

A. H. Klopf, 1972, Brain funtion and adaptive sytems – a heterostatic theory. Teachnical report AFCRL–72–0164, Air Force Cambridge Research Laboratories, Bedford, MA, USA.

A. H. Klopf, 1982, *The Hedonistic Neuron: A Theory of Memory, Learning and Intelligence.* Hemishere, Washington,D.C., USA.

A. H. Klopf, 1988, A neuronal model of classical conditioning. *Psychobiology*, 16:85–125.

R. E. Korf, 1990, Real–time heuristic search. *Aritificial Inteligence*, 42:189–211.

P. R. Kumar, 1985, A survey of some results in stochastic adaptive control. *SIAM Journal of Control and Optimization*, 23:329–380.

L. J. Lin, 1991a, Programming robots using reinforcement learning and teaching. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 781–786, MIT Press, Cambridge, MA, USA.

L. J. Lin, 1991b, Self–improvement based on reinforcement learning, planning and teaching. In *Machine Learning: Proceedings of the Eighth International Workshop*, L. A. Birnbaum and G. C. Collins, editors, pages 323–327, Morgan Kaufmann, San Mateo, CA, USA.

L. J. Lin, 1991c, Self–improving reactive agents: Case studies of reinforcement learning frameworks. In *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behaviour*, pages 297–305, MIT Press, Cambridge, MA, USA.

L. J. Lin, 1992, Self–improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3/4):293–321.

L. J. Lin, 1993, Hierarchical learning of robot skills by reinforcement. In *Proceedings of the 1993 International Conference on Neural Networks*, pages 181–186.

C. S. Lin and H. Kim, 1991, CMAC–based adaptive critic self–learning control. *IEEE Trans. on Neural Networks*, 2(5):530–533.

A. Linden, 1993, On discontinuous Q–functions in reinforcement learning. Available via anonymous ftp from archive.cis.ohio–state.edu in directory /pub/neuroprose.

P. Maes and R. Brooks, 1990, Learning to coordinate behaviour. In *Proceedings of the Eighth National Conferrence on Artificial Intelligence*, pages 796–802, Morgan Kaufmann, San Mateo, CA, USA.

P. Magriel, 1976, *Backgammon.* Times Books, New York, USA.

S. Mahadevan and J. Connell, 1991, Scaling reinforcement learning to robotics by exploiting the subsumption architecture. In *Machine Learning: Proceedings of the Eighth International Workshop*, L. A. Birnbaum and G. C. Collins, editors, pages 328–332, Morgan Kaufmann, San Mateo, CA, USA.

P. Mazzoni, R. A. Andersen, and M. I. Jordan, 1990, A_{R-P} learning applied to a network model of cortical area 7a. In *Proceedings of the 1990 International Joint Conference on Neural Networks*, 2:373–379.

D. Michie and R. A. Chambers, 1968, BOXES: An experiment in adaptive control. *Machine Intelligence 2*, E. Dale and D. Michie, editors, pages 137–152, Oliver and Boyd.

J.D.R. Millan and C. Torras, 1992, A reinforcement connectionist approach to robot path finding in non maze–like environments. *Machine Learning*, 8:363–395.

M. L. Minsky, 1954, *Theory of Neural–Analog Reinforcement Systems and Application to the Brain–Model Problem.* Ph.D. Thesis, Princeton University, Princeton, NJ, USA.

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

A. W. Moore, 1990, *Efficient Memory–based Learning for Robot Control.* Ph.D. Thesis, University of Cambridge, Cambridge, U.K.

A. W. Moore, 1991, Variable resolution dynamic progranmming: Efficiently learning action maps in multivariate real–vlaued state–spaces. In *Machine Learning: Proceedings of the Eighth International Workshop*, L. A. Birnbaum and G. C. Collins, editors, pages 328–332, Morgan Kaufmann, San Mateo, CA, USA.

A. W. Moore and C. G. Atkeson, 1993, Memory–based reinforcement learning: Efficient computation with prioritized sweeping. In *Advances in Neural Information Processing Systems 5*, S. J. Hanson, J. D. Cowan, and C. L. Giles, editors, pages 263–270, Morgan Kaufmann, San Mateo, CA, USA.

M. C. Mozer and J. Bacharach, 1990a, Discovering the structure of reactive environment by exploration. In *Advances in Neural Information Processing 2*, D. S. Touretzky, editor, pages 439–446, Morgan Kaufmann, San Mateo, CA, USA.

M. C. Mozer and J. Bacharach, 1990b, Discovering the structure of reactive environment by exploration. *Neural Computation*, 2:447–457.

K. Narendra and M. A. L. Thathachar, 1989, *Learning Automata: An Introduction.* Prentice Hall, Englewood Cliffs, NJ, USA.

J. Peng and R. J. Williams, 1993, Efficient learning and planning within the Dyna framework. In *Proceedings of the 1993 International Joint Conference on Neural Networks*, pages 168–174.

J. C. Platt, 1991, Learning by combining memorization and gradient descent. *Advances in Neural Information Processing Systems 3*, R. P. Lippmann, J. E. Moody, and D. S. Touretzky, editors, pages 714–720, Morgan Kaufmann, San Mateo, CA, USA.

B. E. Rosen, J. M. Goodwin, and J. J. Vidal, 1991, Adaptive Range Coding. *Advances in Neural Information Processing Systems 3*, R. P. Lippmann, J. E. Moody, and D. S. Touretzky, editors, pages 486–494, Morgan Kaufmann, San Mateo, CA, USA.

S. Ross, 1983, *Introduction to Stochastic Dynamic Programming.* Academic Press, New York, USA.

G. A. Rummery and M. Niranjan, 1994, On–line Q–learning using connectionist systems. Technical Report CUED/F–INFENG/TR 166, University of Cambridge, Cambridge, England.

A. L. Samuel, 1959, Some studies in machine learning using the game of checkers. *IBM Journal on Research and development*, pages 210–229. (Reprinted in *Computers and Thought*, E. A. Feigenbaum and J. Feldman, editors, McGraw–Hill, New York, 1963.)

A. L. Samuel, 1967, Some studies in machine learning using the game of checkers, II – Recent progress. *IBM Journal on Research and Development*, pages 601–617.

O. Selfridge, R. S. Sutton, and A. G. Barto, 1985, Training and tracking in robotics. In *Proceedings of the Ninth International Joint Conference of Artificial Intelligence*, A. Joshi, editor, pages 670–672, Morgan Kaufmann, San Mateo, CA, USA.

J. F. Shepansky and S. A. Macy, 1987, Teaching artificial neural systems to drive: Manual training techniques for autonomous systems. ' In *Proceedings of the First Annual International Conference on Neural Networks*, San Diego, CA, USA.

S. P. Singh, 1991, Transfer of learning across composition of sequential tasks. In *Machine Learning: Proceedings of the Eighth Inernational Workshop*, L. A. Birnbaum and G. C. Collins, editors, pages 348–352, Morgan Kaufmann, San Mateo, CA, USA.

S. P. Singh, 1992a, Reinforcement learning with a hierarchy of abstract models. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, San Jose, CA, USA.

S. P. Singh, 1992b, On the efficient learning of multiple sequential tasks. In *Advances in Neural Information Processing Systems 4*, J. E. Moody, S. J. Hanson, and R. P. Lippmann, editors, pages 251–258, Morgan Kaufmann, San Mateo, CA, USA.

S. P. Singh, 1992c, Scaling Reinforcement learning algorithms by learning variable temporal resolution models. In *Proceedings of the Ninth International Machine Learning Conference.*

S. P. Singh, 1992d, Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, 8(3/4):323–339.

S. P. Singh, A. G. Barto, R. Grupen, and C. Connelly, 1994, Robust reinforcement learning in motion planning. In *Advances in Neural Information Processing Systems 6*, J. D. Cowan, G. Tesauro, and J. Alspector, editors, pages 655–662, Morgan Kaufmann, San Fransisco, CA, USA.

S. P. Singh and R. C. Yee, 1993, An upper bound on the loss from approximate optimal–value functions. Technical Report, University of Massachusetts, Amherst, MA, USA.

R. S. Sutton, 1984, *Temporal Credit Assignment in Reinforcement Learning.* Ph.D. Thesis, Univerity of Massachusetts, Amherst, MA, USA.

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

R. S. Sutton, 1990, Integrated architecture for learning, planning, and reacting based on approximating dyanmic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224, Morgan Kaufmann, San Mateo, CA, USA.

R. S. Sutton, 1991a, Planning by incremental dynamic programming. In *Machine Learning: Proceedings of the Eighth International Workshop*, L. A. Birnbaum and G. C. Collins, editors, pages 353–357, Morgan Kaufmann, San Mateo, CA, USA.

R. S. Sutton, 1991b, Integrated modeling and control based on reinforcement learning and dynamic programming. In *Advances in Neural Information Processing Systems 3*, R. P. Lippmann, J. E. Moody, and D. S. Touretzky, editors, pages 471–478, Morgan Kaufmann, San Mateo, CA, USA.

R. S. Sutton and A. G. Barto, 1981, Toward a modern theory of adaptive networks: Expectation and prediction. *Psychological Review*, 88:135–170.

R. S. Sutton and A. G. Barto, 1987, A temporal–difference model of classical conditioning. In *Proceedings of the Ninth Annual Conference of the Cognitive Science Society*, Erlbaum, Hillsdale, NJ, USA.

R. S. Sutton and A. G. Barto, 1990, Time–derivative models of Pavlovian reinforcement. *Learning and Computational Neuroscience: Foundations of Adaptive Networks*, M. Gabriel and J. Moore, editors, pages 497–537, MIT Press, Cambridge, MA, USA.

R. S. Sutton, A. G. Barto, and R. J. Williams, 1991, Reinforcement learning is direct adaptive optimal control. In *Proceedings of th American Control Conference*, pages 2143–2146, Boston, MA, USA.

R. S. Sutton and S. P. Singh, 1994, On step–size and bias in TD–learning. In *Proceedings of the Eighth Yale Workshop on Adaptive and Learning Systems*, pages 91–96, Yale University, USA.

M. Tan, 1991, Larning a cost–sensitive internal representation for reinforcement learning. In *Machine Learning: Proceedings of the Eighth International Workshop*, L. A. Birnbaum and G. C. Collins, editors, pages 358–362, Morgan Kaufmann, San Mateo, CA, USA.

G. J. Tesauro, 1992, Practical issues in temporal difference learning. *Machine Learning*, 8(3/4):257–278.

C. K. Tham and R. W. Prager, 1994, A modular Q–learning architecture for manipulator task decomposition. In *Machine Learning: Proceedings of the Eleventh Intenational Conference*, W. W. Cohen and H. Hirsh, editors, NJ, Morgan Kaufmann, USA. (Available via gopher from Dept. of Engg., University of Cambridge, Cambridge, England.)

S. B. Thrun, 1986, Efficient exploration in reinforcement learning. Technical report CMU–CS–92–102, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA.

S. B. Thrun, 1993, Exploration and model building in mobile robot domains. In *Proceedings of the 1993 International Conference on Neural Networks.*

S. B. Thrun and K. Muller, 1992, Active exploration in dynamic environments. In *Advances in Neural Information Processing Systems 4*, J. E. Moody, S. J. Hanson, and R. P. Lippmann, editors, Morgan Kaufmann, San Mateo, CA, USA.

S. B. Thrun and A. Schwartz, 1993, Issues in using function approximation for reinforcement learning. In *Proceedings of the Fourth Connectionist Models Summer School*, Lawrence Erlbaum, Hillsdale, NJ, USA.

J. N. Tsitsiklis, 1993, Asynchronous stochastic approximation and Q–learning. Technical Report, LIDS–P–2172, Laboratory for Information and Decision Systems, MIT, Cambridge, MA, USA.

P. E. Utgoff and J. A. Clouse, 1991, Two kinds of training information for evaluation function learning. In *Proceedings of the Ninth Annual Conference on Artificial Intelligence*, pages 596–600, Morgan Kaufmann, San Mateo, CA, USA.

C. J. C. H. Watkins, 1989, *Learning from Dealyed Rewards.* Ph.D. Thesis, Cambridge University, Cambridge, England.

C. J. C. H. Watkins and P. Dayan, 1992, Technical note: Q–learning. *Machine Learning*, 8(3/4):279–292.

P. J. Werbos, 1987, Building and understanding adaptive systems: a statistical/numerical approach to factory automation and brain research. *IEEE Transactions on Systems, Man, and Cybernatics.*

P. J. Werbos, 1988, Generalization of back propagation with application to recurrent gas market model. *Neural Networks* 1:339–356.

P. J. Werbos, 1989, Neural network for control and system identification. In *Proceedings of the 28th Conference on Decision and Control*, pages 260–265, Tampa, FL, USA.

P. J. Werbos, 1990a, Consistency of HDP applied to simple reinforcement learning problems. *Neural Networks*, 3:179–189.

P. J. Werbos, 1990b, A menu of designs for reinforcement learning over time, In *Neural Networks for Control*, W. T. Miller, R. S. Sutton, and P. J. Werbos, editors, pages 67–95, MIT Press, MA, USA.

P. J. Werbos, 1992, Approximate dynamic programming for real–time control and neural modeling. In *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*, D. A. White and D. A. Sofge, editors, pages 493–525, Van Nostrand Reinhold, NY, USA.

S. D. Whitehead, 1991a, A complexity analysis of cooperative mechanisims in reinforcement learning. In *Proceedings of the Ninth Conference on Artificial Intelligence*, pages 607–613, MIT Press, Cambridge, MA, USA.

S. D. Whitehead, 1991b, Complexity and cooperation in Q–learning. In *Machine Learning: Proceedings of the Eighth International Workshop*, L. A. Birnbaum and G. C. Collins, editors, pages 363–367, Morgan Kaufmann, San Mateo, CA, USA.

S. D. Whitehead and D. H. Ballard, 1990, Active perception and reinforcement learning. *Neural Computation*, 2:409–419.

R. J. Williams, 1986, Reinforcement learning in connectionist networks: a mathematical analysis. Technical report ICS 8605, Institute for Cognitive Science, University of California at San Diego, La Jolla, CA, USA.

R. J. Williams, 1987, Reinforcement–learning connectionist systems. Technical report NU–CCS–87–3, College of Computer Science, Northeastern University, Boston, MA, USA.

R. J. Williams and L. C. Baird III, 1990, A mathematical analysis of actor–critic architectures for learning optimal controls through incremental dynamic programming. In *Proceedings of the Sixth Yale Workshop on Adaptive and Learning Systems*, pages 96–101, New Haven, CT, USA.