Solution of Delayed Reinforcement Learning Problems Having Continuous Action Spaces

B. Ravindran
Department of Computer Science and Automation
Indian Institute of Science
Bangalore, India

April, 1996

Contents

Acknowledgements

I would first like to express my extreme gratitude to **Prof. S. Sathiya Keerthi**. Without his unflagging patience and excellent guidance this thesis would never have been completed. He is not only an out-standing teacher and a wonderful advisor, but also a great person to move with. The days I have spent, and will spend, with him will be some of the most cherished in my life.

I would like to thank **Prof. M. A. L. Thathachar** for providing me with some references readily. I would also like to thank **Prof. M. Narasimha Murthy** for providing me access to the Artificial Intelligence Lab. I also thank the CSA office staff (present and Achar) for their smiling faces and ready-to-help nature.

To thank all my friends who made my stay at IISc worthwhile and enjoyable would be a herculian task. I will just mention those who have had a direct impact on this thesis and hope the others will understand :-). First I would like to thank Phani, Prem and Prakash for the numerous discussions we have had on this particular problem and various questions in general. Especially Phani for initially guiding me as to what research should be like. Thanks are also due to Shantaram and Krishna for taking valuble time off to explain to me their results.

To Naga goes the credit of making me work on the implementation of an algorithm for the first time, modifications of which I have been using till the end of the work. Pazhani listened to my woes patiently when ever I wanted to vent my feelings and came up with an useful idea, which though I didn't adopt triggered off other things. Suba listened to many a boring lecture on my problem and gave me a confidence boost whenever I needed it.

Last but not the least come my lab mates: Abhi, Chiru, Krish and Shirish, whose patience has been tested to the limits by me. Whenever I wanted to monopolise the computing power in the lab, they acquiesced to my demands though with a lot of grumbling. :-). Chiru really egged me on to finish my thesis, and Abhi tried to shame me into finishing it. I really admired their unflagging enthusiasm in the absence of success. :-).

There are few others who did not contribute to the thesis directly but were instrumental in my being able to concentrate on my thesis. I would like to thank my one time roommates Navaraj and Srinivas for putting up with my idiosyncrasies; my batch mates and wingmates Jayku, Yuva and Maggi for the wonderful atmosphere they provided; also Yuva for the numerous talks we have had on questions of intelligence, learning, quantum physics and such things; KVS and Barda for the numerous occasions they have helped out with a lot of things; and the Coffee Board for helping me to sleep peacefully on many a day (I *mean* day :-) with the assurance that it was always there.

Abstract

This work concerns the solution of delayed Reinforcement Learning problems having continuous action spaces. The problems associated with continuous action spaces are discussed and various existing algorithms for solving the problem are presented. A extension of Q–learning for solving delayed RL problems having continuous action spaces is proposed which overcomes drawbacks associated with existing methods. Simulation results are presented to demonstrate the working of the proposed algorithm.

Chapter 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). RL problems are modelled usually as follows. We have an agent interacting in a closed loop with an environment (see figure 1.1.1). The agent receives as input the current state of the environment and outputs a suitable action. The environment takes as input the action from the agent and outputs the next state and also a scalar evaluation (reinforcement) of the action. The agent's task is to learn an associative mapping, π, from state space X to the action space A, so as to maximize the reinforcement it receives from the environment. RL problems are very difficult since we have very little feedback from the environment as compared to supervised learning, another popular learning paradigm, in which the environment provides the correct value of $\pi(x)$. Many problems encountered in practice cannot be modelled as supervised learning problems either because $\pi(x)$ is unavailable or is too costly to compute.

One example of a RL problem is the two–armed bandit problem. The agent is required to choose between two actions at a given time step. It is then supplied with a scalar reinforcement $r \in \{0, 1\}$, by the environment. In this case the state space X is taken to be a singleton. The task of the agent is to learn the probabilitites of choosing either action, so as to maximize the reinforcement received from the environment.

In the two–armed bandit problem, the agent receives the reinforcement as soon as it

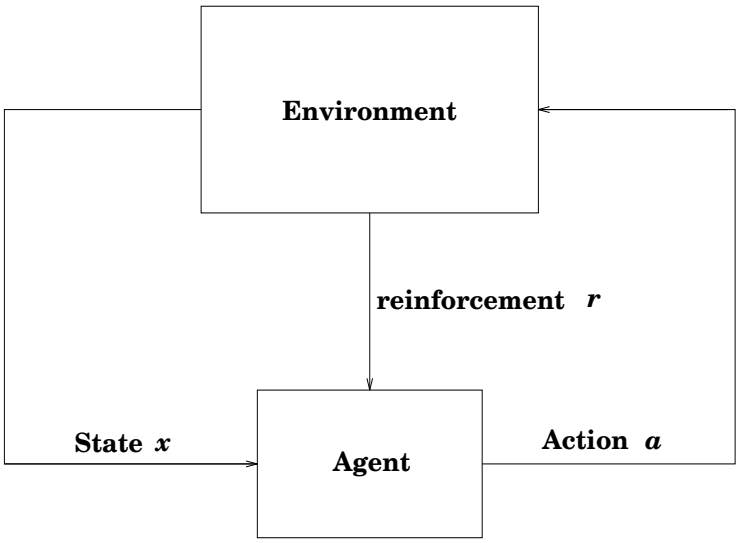

Figure 1.1.1. Model of RL problems

chooses an action. Such problems are known as *Immediate* RL problems in which 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, r, of the (x, a) pair (that indicates how far a is from $\pi(x)$). 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 and are the main focus of this thesis.

Delayed RL problems are much harder to solve than immediate RL problems for the following reason.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.* A typical example of a delayed RL problem is game playing. In game playing the reinforcement available to the agent is the result of the game, say, a 1 for a win and 0 for a loss. The agent does not get any immediate evaluation of its moves but only a cumulative worth of the sequence of moves played by it.

Dynamic Programming (DP) is a well-known tool for solving such problems but it

requires that a complete model of the environment in which the agent is operating be available. Value Iteration and Policy Iteration are two particular iterative DP algorithms that have been popularly used over four decades for off-line solution of delayed RL problems. A model may not be available in many problems, or even if it is available, might be so complex that using it may be infeasible. Delayed RL methods are particularly suited for such situations.

Two such delayed RL methods, namely Actor-critic of Barto, Sutton and Anderson (1983) and Q–learning of Watkins (1989), have made powerful impact on delayed RL research. These algorithms can be interpreted as modification of policy iteration and value iteration respectively.

AI researchers have long been interested in developing game playing machines. Games are well suited for formulating as RL problems and Backgammon is a popular application in which RL methods have proved to be very successful. The *TD–Gammon* program of Tesauro (1995) plays at near grandmaster level, and has performed reasonably well against the top player in the world, losing just a single point. Other successful applications of RL have been in Robot motion planning (Thrun 1993, Mahadevan & Connell 1991), elevator control (Crites & Barto 1996) and process control (Sofge & White 1990).

Most of the RL algorithms developed so far assume that the system operates in a discrete world: discrete state and action spaces and with the system operating in discrete time. In fact, nice convergence results have been established for the discrete case. But most of the typical control problems encountered in practice have continuous state and action spaces and operate in continuous time. With some care, extension of discrete-time delayed RL methods to continuous-time can be easily done (Baird 1993). Also if function approximation techniques are carefully used then the extension of these methods to deal with continuous state spaces is also not difficult. However, extension to continuous action spaces turns out to be non-trivial; this is the problem that we are interested in solving in this thesis.

Little work has been done in tackling problems having a continuous action space. Gullapalli has developed a new stochastic algorithm for immediate RL having continuous action spaces and used it to extend the discrete actor-critic method to continuous action spaces.

Werbos's backpropagated adaptive-critic makes use of a model of the environment and deterministic gradient ascent to operate in continuous action spaces. To overcome the need for a model Brody suggested a modification to it which tries to learn the model on–line. Bradtke has developed a special method of solving Linear Quadratic Regulator problems which have continuous action spaces based on policy iteration and making use of the nice properties of the problem being tackled.

Gullapalli's method is slow because it does stochastic search in the action space using on-line experiences. On the other hand, Werbos' method is fast; however, since it is based on gradient ascent it faces the problem of getting caught in local maxima. Bradtke's method is difficult to extend to general delayed RL problems.

All the above methods are based on policy iteration. We are not aware of any value iteration based method for continuous action spaces. In this thesis we propose a simple idea in this direction. We point out how Q–learning can be extended to continuous action spaces and how the extension overcomes the problems faced by the other three methods mentioned above. To test the working of our method we consider the Linear Quadratic Regulator problem since it enjoys a closed form solution. We devise special function approximators for this problem and demonstrate, by simulations, that our method works well.

This thesis is organized as follows. In the next chapter we present a brief overview of various RL algorithms that are available today and also present a extensive survey of existing litreature. In chapter 3 we discuss in detail the problem of operating with continuous actions and present our algorithm with some simulation results. We conclude with chapter 4 pointing out some directions for further research.

Chapter 2

Survey of Reinforcement Learning Methods

2.1 Introduction

In this chapter we survey some of the existing RL algorithms. As mentioned in the previous chapter, we are interested in delayed RL problems, as these arise in practice more often than immediate RL problems. In this work unless we explicitly state otherwise RL means delayed RL.

Delayed RL algorithms encompass 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 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 chapter we provide a comprehensive 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 chapter, we have obtained

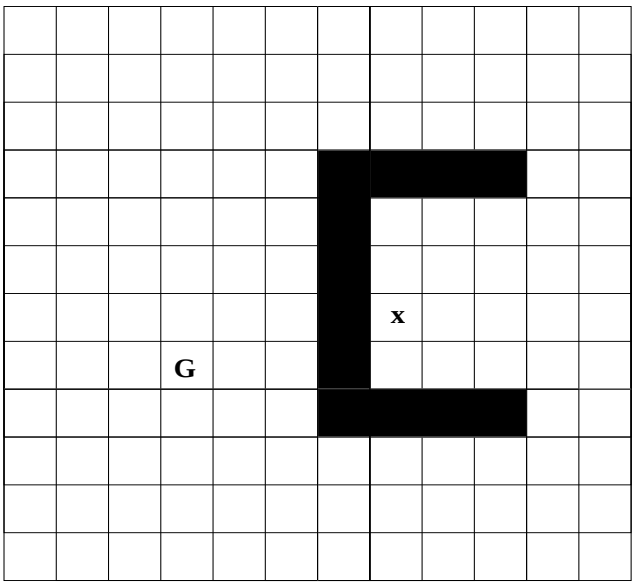

Figure 2.1.1 Navigating in a grid world.

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 2.1.1 Navigating a Robot

Figure 2.1.1 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 2.1.1 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

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

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 section 2.2.

□

Delayed RL problems are much harder to solve than immediate RL problems for the following reason. Suppose, in example 2.1.1 , 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.*

[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 $V^\star(y)$.

Dynamic programming (DP) (Bertsekas 1989; Ross 1983) is a well–known tool for solving problems such as the one in example 2.1.1. 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 this clearly let us return to example 2.1.1 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. *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. 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., neural networks.

[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.

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, $\hat{V}(\cdot; w) : X \to R$, 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

$$\hat{V}(x; w^{\text{new}}) = \hat{V}(x; w) + \beta\eta \tag{2.1.1}$$

For example, in multilayer perceptrons (Hertz *et al* 1991, Haykin 1994) w denotes the set of weights and thresholds in the network and, their updating can be carried out using backpropagation so as to achieve (2.1.1). In the rest of the chapter we will denote the updating process in (2.1.1) as

$$\hat{V}(x; w) := \hat{V}(x; w) + \beta\eta \tag{2.1.2}$$

and refer to it as a *learning rule*.

The chapter is organized as follows. Section 2.2 discusses immediate RL. In section 2.3 we formulate Delayed RL problems and mention some basic results. Methods of estimating total reward are discussed in section 2.4. These methods play an important role in delayed RL algorithms. DP techniques and delayed RL algorithms are presented in section 2.5. Sections 2.6 to 2.8 address various practical issues.

2.2 Immediate Reinforcement Learning

Immediate RL refers to the learning of an associative mapping, $\pi : X \to A$ 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,

$$r(x, \hat{\pi}(x; w)) = r^\star(x) \stackrel{\text{def}}{=} \max_{a \in A} r(x, a) \quad \forall x \in X \tag{2.2.1}$$

Supervised learning is a popular paradigm for learning associative mappings (Hertz *et al* 1991, Haykin 1994). 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 small (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 $r \in [0, 1]$.[4] 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.

[4]Stochatstic Learning Automata algorithms can also be used when X is not a singleton, by employing teams of co–operating automata (Phansalkar & Thathachar 1995, Thathachar & Phansalkar 1995). For more details on such algorithms see Narendra & Thathachar (1989).

One simple way of solving (2.2.1) 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,

$$a = a^k \quad \text{where} \quad z_k = \max_{1 \leq i \leq m} z_i \tag{2.2.2}$$

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

$$g_k(x; w) := g_k(x; w) + \alpha(r(x, a^k) - \hat{r}(x; v)) \tag{2.2.3}$$

where α is a small (positive) step size. If $\hat{r}(\cdot; v) \equiv r^\star$ and (2.2.3) is repeated a number of times for each (x, k) combination, then it should be clear that all non–optimal a^ks will get large negative merit values while an optimal a^k will retain its initial merit value.

Learning $\hat{r}$ requires that all members of A are evaluated by the environment at each x. Clearly, the max selector, (2.2.2) 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 (2.2.2) by a controlled stochastic action selector that generates actions randomly when learning begins and approaches (2.2.2) as learning is completed. A popular stochastic action selector is based on the Boltzmann distribution,

$$p_i(x) \stackrel{\text{def}}{=} \text{Prob}\{a = a^i | x\} = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)} \tag{2.2.4}$$

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 $T \to 0$ the stochastic action selector approaches the max selector, (2.2.2), and, $\tilde{r}(x) \to r^\star(x)$. The ideas here are somewhat similar to those of simulated annealing. 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

$$\hat{r}(x; v) := \hat{r}(x; v) + \beta(r(x, a) - \hat{r}(x; v)) \tag{2.2.5}$$

where β is a small (positive) step size.

Remark Two important comments should be made regarding the convergence of learning rules such as (2.2.5) (we will come across many such learning rules later) which are designed to estimate an expectation by averaging over time.

- Even if $\hat{r}(\cdot; v) \equiv \tilde{r}$, $r(x, a) - \hat{r}(x; v)$ 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 slightly smaller than 1 when learning begins, and then slowly decreased to 0 as learning progresses.

- For good learning to take place, the sequence of x values at which (2.2.5) 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 (2.2.5). 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: $\sigma(T) \to 0$ as $T \to 0$. 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:

$$h(x; w) := h(x; w) + \alpha(r(x, a) - \hat{r}(x; v))\eta \tag{2.2.6}$$

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., to choose σ (Gullapalli 1990).

Gullapalli proposed the *Stochastic Real Valued* (SRV) algorithm as an extension of the Associative Reward-Penalty algorithm. The SRV unit uses two internal parameters θ and ϕ for estimating the action and reward respectively. Let the system be at some state x say and let $\tilde{x}$ be a representation of this state. The output of the unit, y, is generated by a Gaussian distribution having $\mu = \theta^T \tilde{x}$, as the mean. The standard deviation, σ, given by a monotonically decreasing, non–negative function s of $\hat{r} = \phi^T \tilde{x}$, is used to control the amount of exploration. The more the expected reinforcement in that state the the lesser the amount of exploration. The following learning rule is used to update θ:

$$\theta := \theta + \alpha(r(y, x) - \hat{r}) \left(\frac{y - \mu}{\sigma} \right) \tilde{x}$$

ϕ is updated with a rule similar to (2.2.6) as follows:

$$\phi := \phi + \beta(r(y, x) - \hat{r})\tilde{x}$$

Gullapalli (1992a) has also attempted to show that the SRV algorithm belongs to a class of REINFORCE algorithms (Williams 1986, 1987) which have certain convergence properties. Shantaram, Shasty and Thathachar (1994) have given a continuous action set learning automata for stochastic optimization, which combined with ideas of teams of automata can be extended to associative learning tasks. The idea used here is similar to SRV but uses a different scheme to update the parameters μ and σ.

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 $\hat{\pi}$:

$$\hat{\pi}(x;w) := \hat{\pi}(x;w) + \alpha \frac{\partial r(x,a)}{\partial a} \tag{2.2.7}$$

If r is not known, Jordan and Rumelhart suggest that it is learnt using on–line data, and (2.2.7) 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, (2.2.7) 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.

2.3 Delayed Reinforcement Learning

Delayed RL concerns the solution of stochastic optimal control problems. In this section we formulate and discuss the basics of such problems. Solution methods for delayed RL will be presented in section 2.4 and section 2.5. 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. Problems with continuous state and/or action spaces will be dealt with in detail in the next chapter.

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[5] 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*, $V^\pi : X \to R$ as follows:[6]

$$V^\pi(x) = \text{E}\{\sum_{t=0}^{\infty} \gamma^t r(x_t, \pi(x_t)) | x_0 = x\} \tag{2.3.1}$$

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 (2.3.1). 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).

[5] 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). See Jaakkola, Singh and Jordan 1995, and Singh, Jaakkola and Jordan 1994, for a direct treatment of partially observable Markovian decision processes.

[6] Most RL researchers have concerned themselves with the optimization of the expected total discounted reward in (2.3.1). See Heger 1994, for a discussion of an alternative objective function: the minimax criterion.

The expectation in (2.3.1) should be understood as

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

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:

$$V^{\star}(x) = \max_{\pi} V^{\pi}(x) \quad \forall x \tag{2.3.2}$$

$V^{\star}$ 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.

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 (2.3.2). 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 (2.3.2) for every $x \in X$. In other words,

$$V^{\star}(x) = V^{\pi^{\star}}(x)\ , \quad x \in X \tag{2.3.3}$$

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

$$V^{\star}(x) = \max_{a\in A(x)} \left[r(x,a) + \gamma \sum_{y\in X} P_{xy}(a) V^{\star}(y) \right] \tag{2.3.4}$$

The fact that $V^{\star}$ satisfies (2.3.4) 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 (2.3.2); also, if $a = \pi^{x,\star}(x)$ then this term should equal $V^{\star}(x)$. Thus (2.3.4) holds. It also turns out that $V^{\star}$ is the unique function from X to R that satisfies (2.3.4) 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

$$Q^\pi(x,a) = r(x,a) + \gamma \sum_{y \in X} P_{xy}(a) V^\pi(y) \tag{2.3.5}$$

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 (2.3.3) we get

$$V^\star(x) = \max_{a \in A(x)} [Q^\star(x,a)] \tag{2.3.6}$$

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

$$Q^\star(x,a) = r(x,a) + \gamma \sum_{y \in X} P_{xy}(a) \{ \max_{b \in A(y)} Q^\star(y,b) \} \tag{2.3.7}$$

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

$$\pi^\star(x) = \arg\max_{a \in A(x)} [Q^\star(x,a)] \tag{2.3.8}$$

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 section 2.5 for deriving a model–free delayed RL algorithm called Q–Learning (Watkins 1989).

2.4 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*,

$$V^{[n]}(x) = \sum_{\tau=0}^{n-1} \gamma^\tau r_\tau \ , \quad \hat{V}(x; v) = \mathrm{E}(V^{[n]}(x)) \tag{2.4.1}$$

(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

$$\max_x |\hat{V}(x; v) - V(x)| \leq \frac{\gamma^n r_{\max}}{(1-\gamma)} \tag{2.4.2}$$

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

But (2.4.1) 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

$$\hat{V}(x; v) = V^{[n]}(x) \tag{2.4.3}$$

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, (2.4.3) 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 (2.4.3) 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 (2.2.5)):

$$\hat{V}(x;v) := \hat{V}(x;v) + \beta\left[V^{[n]}(x) - \hat{V}(x;v)\right] \tag{2.4.4}$$

where β is a small (positive) step size. Learning can begin with a random choice of v. Eventually, after a number of trials, we expect the $\hat{V}$ resulting from (2.4.4) to satisfy (2.4.2).

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,*

$$V^{(n)}(x) = \sum_{\tau=0}^{n-1} \gamma^{\tau} r_{\tau} + \gamma^{n}\hat{V}(x_n;v) \tag{2.4.5}$$

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,

$$\hat{V}(x;v) := E(V^{(n)}(x)) \quad \forall\, x \tag{2.4.6}$$

Suppose v gets modified to v_{new} in the process of satisfying (2.4.6). Then, similar to (2.4.2) we can easily derive

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

Thus, as we go through a number of learning steps we achieve $\hat{V}\rightarrow V$. 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., (2.4.1), is only guaranteed to achieve the bound in (2.4.2). *Therefore, when a system model is available it is best to choose a small* n*, say* $n = 1$*, and employ (2.4.6).*

Now suppose that, either a model is unavailable or (2.4.6) 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:

$$\hat{V}(x;v) := \hat{V}(x;v) + \beta\left[V^{(n)}(x) - \hat{V}(x;v)\right] \tag{2.4.7}$$

Which is better: (2.4.4) or (2.4.7)? There are two reasons as to why (2.4.7) is better.

[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.

- 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 (2.4.7) 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 (2.4.4) 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 (2.4.4) and (2.4.7) will yield similar performance.

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 $\{V^{(n)}(x) : n \geq 1\}$:

$$V^{\lambda}(x) = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} V^{(n)}(x) \tag{2.4.8}$$

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 (2.4.8) using (2.4.5) we get

$$\begin{aligned} V^{\lambda}(x) &= (1-\lambda)\left[V^{(1)}(x) + \lambda V^{(2)}(x) + \lambda^2 V^{(3)}(x) + \cdots\right] \\ &= r_0 + \gamma(1-\lambda)\hat{V}(x_1; v) \ + \\ &\quad \gamma\lambda\left[r_1 + \gamma(1-\lambda)\hat{V}(x_2; v) \ + \right. \\ &\quad\quad \left. \gamma\lambda\left[r_2 + \gamma(1-\lambda)\hat{V}(x_3; v) \ + \right.\right. \\ &\quad\quad\quad \left.\left. \cdots \right.\right. \end{aligned} \tag{2.4.9}$$

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

$$V^{\lambda}(x) = r(x, \pi(x)) + \gamma(1-\lambda)\hat{V}(x_1; v) + \gamma\lambda V^{\lambda}(x_1) \tag{2.4.10}$$

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[8] 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 $V^{(n)}$:

$$\hat{V}(x;v) := \hat{V}(x;v) + \beta\left[V^\lambda(x) - \hat{V}(x;v)\right] \tag{2.4.11}$$

Define the *temporal difference* operator, Δ by

$$\Delta(x) = r(x,\pi(x)) + \gamma\hat{V}(x_1;v) - \hat{V}(x;v) \tag{2.4.12}$$

$\Delta(x)$ 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 (2.4.9) yields

$$V^\lambda(x) - \hat{V}(x;v) = \Delta(x) + (\gamma\lambda)\Delta(x_1) + (\gamma\lambda)^2\Delta(x_2) + \cdots \tag{2.4.13}$$

Even (2.4.13) is not in a form suitable for use in (2.4.11) 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 (2.4.13) to include only the first N terms, and apply (2.4.11) at $\tau = N+1$, i.e., $(N+1)$ time steps after x occured. However, a simpler and approximate way of achieving (2.4.13) 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

[8]For example, if the underlying dynamic system is deterministic then a value of λ close to 1 is appropriate; on the other hand, if the system is highly stochastic then a value of λ near zero is better.

$(t+1)$, 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 (2.4.13) 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 (2.4.11).

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:

$$\hat{V}(x;v) := \hat{V}(x;v) + \beta e(x,t)\Delta(x_t) \;\; \forall\, x \tag{2.4.14}$$

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

$$e(x,t) = \begin{cases} 0 & \text{if } x \text{ has never been visited} \\ \gamma\lambda e(x,t-1) & \text{if } x_t \neq x \\ 1+\gamma\lambda e(x,t-1) & \text{if } x_t = x \end{cases} \tag{2.4.15}$$

Two important remarks must be made regarding this implementation scheme.

- Whereas the previous learning rules (e.g., (2.4.4), (2.4.7) and (2.4.11)) update $\hat{V}$ only for one x at a time step, (2.4.14) updates the $\hat{V}$ of all states with positive eligibility trace, at a time step. Rule (2.4.14) is suitable for neural hardware implementation, but not so for implementations on sequential computers. In that case one of the following ideas can be tried.

  1. 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 $k = 1$.
  2. The following idea is due to Cichosz (1995). Choose a nonnegative integer m depending on the decay rate $\gamma\lambda$) and truncate the right hand side of (2.4.13) to keep only the first $(m+1)$ terms and get

     $$\hat{V}(x;v) := \hat{V}(x;v) + \beta\, \delta(x)$$

     where

     $$\delta(x) = \Delta(x) + (\gamma\lambda)\Delta(x_1) + \cdots + (\gamma\lambda)^m\Delta(x_m)$$

Thus, if x is the state occuring at time step t, $\hat{V}(x;v)$ gets updated at the end of time step $(t+m)$ and, *more importantly, x is the only state for which $\hat{V}$ is updated at time step $(t+m)$.* The recursion,

$$\delta(x_1) = [\delta(x) - \Delta(x)]/(\gamma\lambda) + (\gamma\lambda)^m \Delta(x_{m+1})$$

can be employed so that the δ computation can be done in constant time even if m is large. Cichosz (1995) has also suggested (with good justification) another update rule based on truncation which is even better than the idea described above.

- The rule for updating eligibility traces, (2.4.15) 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 (2.2.5) applies as well to the learning rules, (2.4.4), (2.4.7), (2.4.11) and, (2.4.14). Dayan and Sejnowski (1993), and Jaakkola *et al* (1994) have shown that, if the real time $TD(\lambda)$ learning rule, (2.4.14) is used, then under appropriate assumptions on the variation of β in time, as $t \to \infty$, $\hat{V}$ 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 (2.4.10), (2.4.12) etc. should be taken as instances generated by the stochastic policy.

Let us now come to the estimation of Q^π. Recall from (2.3.5) 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 (2.4.14):

$$\hat{Q}(x,a;v) := \hat{Q}(x,a;v) + \beta e_Q(x,a,t)\Delta_Q(x_t,a_t) \;\; \forall\,(x,a) \tag{2.4.16}$$

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

$$\Delta_Q(x,a) = r(x,a) + \gamma\hat{Q}(x_1,\pi(x_1);v) - \hat{Q}(x,a;v); \tag{2.4.17}$$

and

$$e_Q(x,a,t) = \begin{cases} 0 & \text{if } (x,a) \text{ has never been visited} \\ \gamma\lambda e_Q(x,a,t-1) & \text{if } (x_t,a_t) \neq (x,a) \\ 1+\gamma\lambda e_Q(x,a,t-1) & \text{if } (x_t,a_t) = (x,a) \end{cases} \tag{2.4.18}$$

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 (2.4.17) by

$$\Delta_Q(x,a) = r(x,a) + \gamma\tilde{V}(x_1) - \hat{Q}(x,a;v) \tag{2.4.19}$$

where

$$\tilde{V}(x_1) = \sum_{b\in A(x_1)} \text{Prob}\{\pi(x)=b\}\hat{Q}(x_1,b;v) \tag{2.4.20}$$

Rummery and Niranjan (1994) suggest the use of (2.4.17) even if π is stochastic; in that case, the $\pi(x_1)$ in (2.4.17) 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 (2.4.17), and so it fits-in better with the definition of Q^π in (2.3.5). Also, if the the size of $A(x_1)$ is small then the computation of $\tilde{V}(x_1)$ is not much more expensive than that of $\hat{Q}(x_1,\pi(x_1);v)$.

2.5 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 section 2.3. Since delayed RL methods also solve the same problem, how do they differ from DP methods?[9] Following are the main differences.

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

- 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.

- 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.

2.5.1 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.

2.5.1.1 Value Iteration

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

$$V^\star(x) = \lim_{n\to\infty} V_n^\star(x) \tag{2.5.1}$$

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:

$$V_1^\star(x) = \max_{a\in A(x)} r(x,a) \;\; \forall\, x \tag{2.5.2}$$

Then, the recursion,[10]

$$V_{n+1}^\star(x) = \max_{a\in A(x)} \left[r(x,a) + \gamma \sum_y P_{xy}(a) V_n^\star(y) \right] \;\; \forall\, x \tag{2.5.3}$$

can be used to compute $V_{n+1}^\star$ for $n = 1, 2, \cdots$. (Iterations can be terminated after a large number (N) of iterations, and $V_N^\star$ 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, $V : X \to R$ is a policy, π that is *greedy with respect to* V, i.e.,

$$\pi(x) = \arg \max_{a\in A(x)} \left[r(x,a) + \gamma \sum_y P_{xy}(a) V(y) \right] \;\; \forall\, x \tag{2.5.4}$$

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.*

[10] One can also view the recursion as doing a fixed-point iteration to solve Bellman's optimality equation, (2.3.4).

In the above, we have assumed that (2.5.1) 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* $V_1^\star$ = *an arbitrary function over states.*

Repeat

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

$$V_{n+1}^\star(x) = \begin{cases} \max_{a \in A(x)} \left[r(x,a) + \gamma \sum_y P_{xy}(a) V_n^\star(y) \right] & \text{if} \quad x \in B_n \\ V_n^\star(x) & \text{otherwise} \end{cases} \tag{2.5.5}$$

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

If we choose $V_1^\star$ as in (2.5.2) 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 $M_n = \max_x |V_n^\star(x) - V^\star(x)|$. Then $\max_{x \in B_n} |V_{n+1}^\star(x) - V^\star(x)| \leq \gamma M_n$.

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

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

Similarly,

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

and so the theorem is proved. □

The theorem implies that $M_{n+1} \leq M_n$ 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 $M_{k+K} \leq \gamma M_k$. Therefore, *if the value of each state is backed up infinitely often, then (3.5.1) holds.*[11] In the case of value iteration, the value of each state is backed up at each iteration and so (2.5.1) 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, which is simple, is something that we have stressed earlier. In most problems (e.g., playing games such as checkers and backgammon)

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

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 be backed up infinitely often[12] 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[13]: (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 (2.5.5). Convergence of such adaptive methods has been proved by Gullapalli and Barto (1994).

2.5.1.2 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

$$V^\mu(x) \geq V^\pi(x) \ \ \forall\, x \tag{2.5.6}$$

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

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

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

Policy Improvement Theorem

The policy μ is uniformly better than policy π if

$$Q^{\pi}(x, \mu(x)) \geq V^{\pi}(x) \ \ \forall\, x \tag{2.5.7}$$

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. □

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

$$\mu(x) = \arg \max_{a \in A(x)} Q^{\pi}(x, a) \ \ \forall\, x \tag{2.5.8}$$

then (2.5.7) 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 (2.3.5).*
2. *Find μ using (2.5.8) and compute V^{μ}.*
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 section 2.5.2.1 we will discuss a well-known model–free method called the *actor-critic* method which gives an inexpensive approximate way of implementing policy iteration.

2.5.2 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.

2.5.2.1 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[14] is best derived by combining the ideas of section 2.2 and section 2.4 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 (2.2.4).[15]

[14] A mathematical analysis of this method has been done by Williams and Baird (1993a).

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

Critic (V^π) Maintain an approximator, $\hat{V}(\cdot; w) : X \to R$ that estimates the value function (expected total reward) corresponding to the stochastic policy mentioned above. The ideas of section 2.4 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 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 (2.2.3) would be

$$g_k(x; w) := g_k(x; w) + \alpha \left[\rho(x, a^k) - \hat{V}(x; v)\right] \tag{2.5.9}$$

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

$$\rho(x, a^k) \approx r(x, a^k) + \gamma \sum_y P_{xy}(a^k)\hat{V}(y; v) \tag{2.5.10}$$

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

$$\rho(x, a^k) \approx r(x, a^k) + \gamma \hat{V}(x_1; v) \tag{2.5.11}$$

where x_1 is the state occuring in the real time operation when action a^k is applied at state x. Since the right hand side of (2.5.11) is an unbiased estimate of the right hand side of (2.5.10), using this approximation in the averaging learning rule (2.5.9) will not lead to errors. Using (2.5.11) in (2.5.9) gives

$$g_k(x; w) := g_k(x; w) + \alpha\Delta(x) \tag{2.5.12}$$

where Δ is as defined in (2.4.12). 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 (2.2.4) 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 (2.5.12) 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, (2.4.14) 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 (2.4.11) for all states visited in the trial, and then update g using (2.5.12) for all (state,action) pairs experienced in the trial.

2.5.2.2 Q–Learning

Just as the actor–critic method is a model-free, on-line way of approximately implementing policy iteration, Watkins' Q–Learning (Watkins 1989) 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 (2.5.5). Q–Learning overcomes this problem elegantly by operating with the Q–function instead of the value function. (Recall, from section 2.3, 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, (2.3.7), 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 section 2.5.1.1 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 section 2.5.1.1 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

$$\hat{Q}(x,a;v) := r(x,a) + \gamma \sum_{y} P_{xy}(a) \max_{b \in A(y)} \hat{Q}(y,b;v) \tag{2.5.13}$$

Using this rule, all the ideas of section 2.5.1.1 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 (2.5.13) 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 (2.5.13) via the "averaging" learning rule,

$$\hat{Q}(x,a;v) := \hat{Q}(x,a;v) + \beta \left[r(x,a) + \gamma \max_{b \in A(x_1)} \hat{Q}(x_1,b;v) - \hat{Q}(x,a;v) \right] \tag{2.5.14}$$

If (2.5.14) is carried out we say that the Q–value of (x,a) has been backed up. Using (2.5.14) 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 (2.5.14).*
3. *Reset $x := x_1$.*

The remark made below equation, (2.2.5) in section 2.2 is very appropriate for the learning rule, (2.5.14). Watkins showed[16] 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 actions;[17] (2) doing

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

[17] 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.

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 section 2.4. 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, (2.5.14) should match (note the use of π in (2.4.17) and (2.4.20)). Thus $TD(\lambda)$ can be used if we employ the greedy policy,

$$\pi(x) = \arg \max_{a \in A(x)} \hat{Q}(x, a; v) \tag{2.5.15}$$

in step 1.[18] [19] 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),

$$\text{Prob}\{\pi(x) = a|x\} = \frac{exp(\hat{Q}(x, a; v)/T)}{\sum_{b \in A(x)} exp(\hat{Q}(x, b; v)/T)}, \quad a \in A(x) \tag{2.5.16}$$

where $T \in [0, \infty)$. When $T \to \infty$ all actions have equal probabilities and, when $T \to 0$ the stochastic policy tends towards the greedy policy in (2.5.15). 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, (2.4.19) estimates expected returns for the policy at each T, and, as $T \to 0$, $\hat{Q}$ will converge to $Q^\star$.

An important remark needs to be made regarding the application of Q–Learning to RL problems which result from the time-discretization of continuous-time problems. As the

[18] Although the greedy policy defined by (2.5.15) keeps changing during a trial, the $TD(\lambda)$ estimate can still be used because $\hat{Q}$ is varied slowly.

[19] If more than one action attains the maximum in (2.5.15) then for convenience we take π to be a stochastic policy that makes all such maximizing actions equally probable.

discretization time period goes to zero it turns out that the Q function tends to become a constant for all a for a given x and hence it is unsuitable to use Q–Learning for continuous-time problems. For such problems Baird (1993) has suggested the use of an appropriate modification of the Q function called the Advantage function. See Harmon *et al* (1995) also.

2.6 Function–Approximators in RL

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). Almost 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.

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, neural 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)

[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.

have reported bad results. A modified form of Platt's *Resource Allocation Network* (RAN) (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–neural 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.[21] 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. Sutton (1996), in an effort to study this problem, has used CMACs successfully in problems where MLPs have been reported to have failed (Boyen & Moore 1995).

When methods based on clustering are employed for function approximation it would be helpful to know where to put the clusters. CMACs and RBF based methods start with

[21]The criterion for 'nearness' must be chosen properly depending on the problem being solved. For instance, in example 2.1.1 (see figure 2.1.1) two states on opposite sides of the barrier but whose coordinate vectors are near, have vastly different optimal 'cost-to-go' values. Hence the function approximator should not generalize the value at one of these states using the value at the other. Dayan (1993) gives a general approach for choosing a suitable 'nearness' criterion so as to improve generalization.

a fixed number of clusters and though one can change the position and size of the clusters, they do not perform well when the nature of the problem is such that many small clusters are needed in some regions and few large ones at most other places, like in example 2.1.1. One way of overcoming this problem would be to use ontogenic networks in which the number and position of clusters is varied dynamically and clusters are added whenever and wherever they are needed. One such ontogenic algorithm is RAN. Anderson has used a variation of RANs in which he fixes the number of clusters. The clusters are dynamically deleted and added as needed, keeping the number of clusters constant. We have done some preliminary investigation into the use of a more direct RAN based method on the problem presented in example 2.1.1. We found that RANs performed much better than a RBF network with a fixed number of nodes (comparable to the number of nodes inserted typically by a RAN) distributed uniformly over the entire input space.

The effect of errors introduced by function approximators on the optimal performance of the controller has not been well understood.[22] It has been pointed out by Watkins (1989), Bradtke (1993), Bertsekas (1994) 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.

Mance Harmon of Wright-Patterson Air Force Base, Ohio, has pointed out the following explanation as to why function approximators used with RL have difficulties. The generalization that takes place when updating the approximation systems can, as a side effect, change the target value. For instance, when the update rule (2.4.14), which is based on $\Delta(x_t)$, is performed, the resulting change in $\hat{V}$ together with generalization can lead to a sizeable change in $\Delta(x_t)$. We are then, in effect, shooting at a moving target. This is a cause of instability, and the propensity of the weights, in many cases, to grow to infinity.

[22] Bertsekas(1989), Singh and Yee (1993), and Williams and Baird (1993b) have derived some general theoretical bounds for errors in value function in terms of function approximator error. Tsitsiklis and Van Roy (1994) have derived bounds for errors when feature-based function approximators are used.

To overcome this problem Baird and Harmon (1993) have suggested a residual gradient approach in which gradient descent is performed on the mean square of residuals such as $\Delta(x_t)$. Then one can expect convergence in a way similar to how convergence takes place in the backpropagation algorithm. A similar approach has also been suggested by Werbos (1987).

Overall, it must be mentioned that much work needs to be done on the use of function approximators for RL, and clear guidelines are yet to emerge.

2.7 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 neural 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 decom-

position 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[23], 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.

2.8 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

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

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).

2.9 Conclusion

In this chapter we have given 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.

Chapter 3

RL for Continuous Action Spaces

3.1 Introduction

Typical optimal control problems involve continuous state and action spaces. It is easy to extend the algorithms discussed in the previous chapter to continuous state spaces by the use of appropriate function approximators that generalize a real-time experience at a state to all topologically nearby states. Many such function approximators and the caveats in using them have been discussed in section 2.6. But the extension of existing algorithms to continuous actions is difficult. There are two difficulties associated with this. If one tries to extend, say, Q–learning to continuous action spaces then the following difficulties are immediately obvious:

1. The max operation in (2.5.14) is now difficult.

2. Defining a policy in terms of the value function is also non-trivial since that too needs the max operation. (see (2.5.15)).

Hence some special ideas are needed for operating with continuous action spaces. All the existing algorithms take care of these problems by adopting sutiable techniques.

In this chapter we survey the existing model-free and model-based methods for continuous action spaces. We present a new algorithm which is an extension of Q–learning to continuous action spaces. We also present a modification of the algorithm applicable in cases where the $Q(x, a)$ function is unimodal in a for each fixed x. This, to the best of our

knowledge, is the first attempt to extend value-iteration based methods like Q–learning to continuous action spaces. We then present simulation results to show that our algorithm works.

3.2 Existing methods

In this section we present some of the existing methods of handling continuous action spaces. Just to make the presentation easy, we will make the assumption that the system being controlled is deterministic. Let

$$x_{t+1} = f(x_t, a_t) \tag{3.2.1}$$

describe the transition.

3.2.1 A Model-based Method: The Back-propagated Adaptive Critic

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 described in the previous chapter. 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, (2.4.14) of section 2.4. The actor tries to improve the policy at each time step using the hint provided by the policy improvement theorem in (2.5.7). To be more specific, let us define

$$Q(x, a) \stackrel{\text{def}}{=} r(x, a) + \gamma \hat{V}(f(x, a); v) \tag{3.2.2}$$

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 (3.2.1). 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 (2.5.7), we aim to adjust $\hat{\pi}(x_t; w)$ to give a new value, a^{new} such that

$$Q(x_t, a^{\text{new}}) > Q(x_t, a_t) \tag{3.2.3}$$

For a control problem in which Q is differentiable and there are no action constraints, a simple learning rule that achieves this requirement is

$$\hat{\pi}(x_t; w) := \hat{\pi}(x_t; w) + \alpha \frac{\partial Q(x_t, a)}{\partial a}|_{a=a_t} \tag{3.2.4}$$

where α is a small (positive) step size. The partial derivative in (3.2.4) can be evaluated using

$$\frac{\partial Q(x_t, a)}{\partial a} = \frac{\partial r(x_t, a)}{\partial a} + \gamma \frac{\partial \hat{V}(y; v)}{\partial y}|_{y=f(x_t,a)} \frac{\partial f(x_t, a)}{\partial a} \tag{3.2.5}$$

3.2.2 Model–free methods

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[1] 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 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.

3.2.3 SRV–based algorithm

A more direct model–free method can be derived using the ideas of actor–critic method of section 2.5.2.1. One difficulty in extending those ideas to continuous aciton spaces is that we now cannot maintain a different function approximator to give the value of each action involved. (See (2.5.9).) We can overcome this by employing ideas from section 2.2. We can use a learning rule similar to the SRV algorithm for adapting $\hat{\pi}$. This method was proposed and successfully demonstrated by Gullapalli on some practical problems (Gullapalli 1992a; Gullapalli *et al* 1994).

[1] This deficiency was also pointed out by Gullapalli (1992b).

3.2.4 Bradtke's Policy Iteration Scheme based on Q–functions

Bradtke (1993) has chosen a special problem, Linear Quadratic Regulation (LQR), and has developed a modification of the Q–learning algorithm which is close to policy iteration and is applicable to this specific problem. Instead of trying to learn the optimal Q function right away using the learning rule in (2.5.14), which he calls the *optimizing Q–learning rule*, Bradtke first fixes a policy, π, and then learns the Q function corresponding to that policy using the following rule:

$$\hat{Q}(x,a;v) := \hat{Q}(x,a;v) + \beta \left[r(x,a) + \gamma \hat{Q}(x_1, \pi(x_1); v) - \hat{Q}(x,a;v) \right] \tag{3.2.6}$$

After the Q function for that policy is learnt sufficiently well he performs a policy improvement step so as to arrive at a better policy. This is done by making use of some nice properties of the LQR problem. He has also shown that, theoretically, this algorithm converges to the optimal Q function when applied to the LQR problem. It should be mentioned that this is the only theoretical convergence result that has been established thus far for delayed RL problems involving continuous action spaces.

3.3 Extension of Q–learning to Continuous Action Spaces

In this section we propose a new model–free method based on Q–learning. The optimizimg Q–learning rule introduced in section 2.5.2.2 is as follows:

$$\hat{Q}(x_t,a_t;v) := \hat{Q}(x_t,a_t;v) + \beta \left[r(x_t,a_t) + \gamma \max_{b \in A(x_{t+1})} \hat{Q}(x_{t+1},b;v) - \hat{Q}(x_t,a_t;v) \right] \tag{3.3.1}$$

As mentioned in the beginning of this chapter the max operation in the above equation is non-trivial in case of continuous action spaces. In order to overcome this difficulty we maintain another function approximator, $\hat{\pi}$, that learns the action corresponding to the best $\hat{Q}$ value for a given state x. In other words, $\hat{\pi}$ is a policy network that learns a *policy that is optimal with respect to* $\hat{Q}$. We now employ the learning rule:

$$\hat{Q}(x_t,a_t;v) := \hat{Q}(x_t,a_t;v) + \beta \left[r(x_t,a_t) + \gamma \hat{Q}(x_{t+1}, \hat{\pi}(x_{t+1}); v) - \hat{Q}(x_t,a_t;v) \right] \tag{3.3.2}$$

Let us now see how $\hat{\pi}$ can be adapted. The aim is to have

$$\hat{\pi}(x_t;w) = a^\star$$

where,

$$a^\star \stackrel{\text{def}}{=} \arg \max_{a \in A(x_t)} \hat{Q}(x_t, a; v)$$

A simple **general scheme** is to actually compute $a^\star$ using a global optimization algorithm and then adapt w so that $\hat{\pi}(x_t, w)$ moves towards $a^\star$. Note that, the running of the global optimization algorithm uses $\hat{Q}$ only and does not require any on-line experiences from the environment. Given the availability of fast and inexpensive processors and the fact that the dimension of a is usually small in most control applications, solving a global optimization problem in the action space within one discretization period of the control system is computationally feasible. This is especially true in the case of chemical process control systems for which the discretization period is in the order of minutes.

There is an interesting and useful class of problems (see section 3.5.1) for which the optimal Q function is unimodal (e.g. concave) in a for each fixed x. For such cases, for each fixed x, a local maximum of the Q function is also a global maximum and hence a **gradient ascent scheme** can be used to learn $\hat{\pi}$:

$$\hat{\pi}(x_t; w) := \hat{\pi}(x_t; w) + \alpha \frac{\partial \hat{Q}(x_t, a)}{\partial a}|_{a=a_t} \tag{3.3.3}$$

The partial derivative can be easily computed if we employ a suitable function approximator. We employ connectionist networks in this work. As shown in figure 3.3.1, the output of the $\hat{\pi}$ network acts as an input to the $\hat{Q}$ network. The derivative of the output of the $\hat{Q}$ network with respect to one of its inputs, namely a, is easy to compute by techniques such as back-propagation (Haykin 1994).

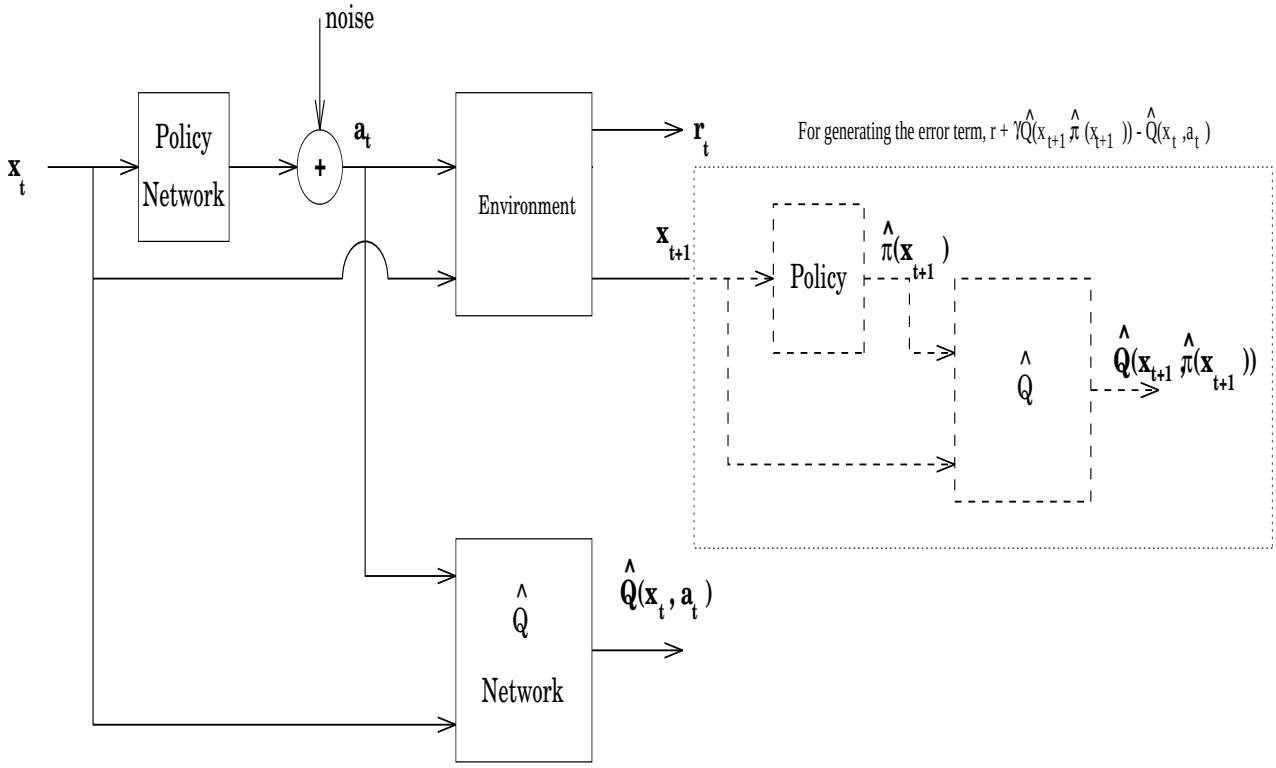

Figure 3.3.1 Q–learning in continuous action spaces

Learning Trial

Set $t = 0$.

Initialize the $\hat{Q}$ *and* $\hat{\pi}$ *networks.*

Repeat (for a number of time steps)

1. *Choose input* x_t
2. *compute* a_t *according to an exploration policy*[2] *and apply it to the system. Let* x_{t+1} *be the resulting state.*
3. *compute* $q_t = \hat{Q}(x_t, a_t)$.
4. *Update* $\hat{Q}$ *using (3.3.3).*
5. *update* $\hat{\pi}$ *using (3.3.2)*
6. *set* $t = t + 1$

3.4 Comparison with Earlier Works

In this section we compare our work with the earlier attempts to tackle problems with continuous action spaces. First let us consider Werbos' Back-propagated adaptive-critic. It converges to the solution quickly since a gradient ascent is performed (see (3.2.4)). The drawback of doing this is that gradient ascent suffers from local maxima problems and might not converge to the optimal values if local maxima exist.

Gullapalli's SRV–based algorithm overcomes the local maxima problem since it does a stochastic search of the a space. This means that a lot of directions have to be tried out in

[2]The exploration policy can be different from $\hat{\pi}$; however, if $TD(\lambda)$ updates are to be performed for faster learning then the exploration policy should be the same as $\hat{\pi}$. See the comments made after (2.5.15).

the a space and everytime a new direction is to be tried out it needs an on-line experience. This makes the algorithm very slow.

Bradtke's Policy iteration scheme based on Q functions does not suffer from these drawbacks, but is specialized only to the LQR problem and hence lacks general applicability.

In our general scheme we use a global optimization technique such as simulated annealing. We employ the gradient ascent scheme only when $Q(x, a)$ is unimodal in a for a given x. This too is guaranteed to find a global maximum. So both the schemes do not suffer from the problem of local maxima. Also the maximization step uses only $\hat{Q}$ and does not need any expensive on-line experience. Hence our algorithm is faster than schemes which need on-line experience. We have not made any special assumptions about the problem in arriving at our algorithm. Hence our algorithm is not restricted to any special class of problems.

All the existing schemes for solving delayed RL problems with continuous action spaces use policy iteration based methods. We have extended Q–learning, a value iteration based method, to continuous action spaces. This is the first attempt at employing a value iteration based method to solve delayed RL problems having continuous action spaces.

3.5 Testing

Most applications involving continuous action spaces come from the design of control systems. Many such interesting problems have linear dynamics. Even when dynamics are non-linear, linearization techniques are often used to obtain excellent linear approximations. Thus it is useful to consider the application of our algorithm to optimal control problems having linear dynamics. Further, closed form solutions are available for some of these problems; for such problems it is easy to check if our learning method yields correct solutions.

3.5.1 Linear Regulation Problem

Consider the deterministic, linear, time-invariant, dynamical system given by:

$$x_{t+1} = Ax_t + Ba_t$$

where A and B are matrices of dimensions $n \times n$ and $n \times m$ respectively. Let r be a concave function[3] of (x,a). For such problems the following can be easily shown:

Lemma: $V^\star$ and $Q^\star$ are concave functions.

Proof: Let $\overline{x}$ and $\tilde{x}$ be two states, and, $\{(\overline{x}_t, \overline{a}_t)\}$ and $\{(\tilde{x}_t, \tilde{a}_t)\}$ be the state-action sequences generated by the optimal policy starting from $\overline{x}_0 = \overline{x}$ and $\tilde{x}_0 = \tilde{x}$ respectively. Let $0 \leq \delta \leq 1$, $x = \delta\overline{x} + (1-\delta)\tilde{x}$, $x_t = \delta\overline{x}_t + (1-\delta)\tilde{x}_t$ and $a_t = \delta\overline{a}_t + (1-\delta)\tilde{a}_t$. Clearly, $x_0 = x$. Then by the optimality of $V^\star(x)$ and the concavity of r, we get:

$$\begin{aligned} V^\star(x) &\geq \textstyle\sum_t \gamma^t r(x_t, a_t) \\ &= \textstyle\sum_t \gamma^t r(\delta\overline{x}_t + (1-\delta)\tilde{x}_t, \delta\overline{a}_t + (1-\delta)\tilde{a}_t) \\ &\geq \textstyle\sum_t \gamma^t \left[\delta r(\overline{x}_t, \overline{a}_t) + (1-\delta) r(\tilde{x}_t, \tilde{a}_t)\right] \\ &= \delta V^\star(\overline{x}) + (1-\delta) V^\star(\tilde{x}) \end{aligned}$$

A similar proof shows that $Q^\star$ is also a concave function. □

Consider the case where the cost at every time step is a quadratic function of the state and the control signal: $-r_t = x_t^T E x_t + a_t^T F a_t$, where E is a symmetric positive semi-definite matrix of dimension $n \times n$ and F is a symmetric positive definite matrix of dimension $m \times m$. This now becomes a Linear Quadratic Regulation (LQR) problem.

The value function $V^\pi(x_t)$ is defined in the usual way as the discounted sum of all costs that will be incurred by using π for all times from t onward. From Linear-quadratic control theory we know that $\pi^\star$ is a linear policy, i.e., $\pi^\star(x) = U^\star x$ for some $m \times n$ matrix $U^\star$. Hence it is sufficient if we optimize over all linear policies: $\pi(x) = Ux$. For a linear policy V^π is a quadratic function of the states and can be expressed as $V^\pi(x) = -x^T P^\pi x$, where P^π is a $n \times n$ symmetric positive definite matrix. Let $P^\star$ denote $P^{\pi^\star}$.

$P^\star$ is given by the solution of the Ricatti equation:

$$P = E - A^T PB(F + B^T PB)^{-1} B^T PA + A^T PA$$

and $Q^\star$ is given by:

$$Q^\star(x,a) = -\begin{pmatrix} x^T & a^T \end{pmatrix} \begin{pmatrix} E + A^T P^\star A & A^T P^\star B \\ B^T P^\star A & F + B^T P^\star B \end{pmatrix} \begin{pmatrix} x \\ a \end{pmatrix} \tag{3.5.1}$$

[3] A function $f : R^k \to R$ is said to be concave if: $f(\delta\overline{y} + (1-\delta)\tilde{y}) \geq \delta f(\overline{y}) + (1-\delta) f(\tilde{y}) \;\; \forall \overline{y}, \tilde{y} \in r^k, \; 0 \leq \delta \leq 1$

3.5.2 Representation of Functions for the LQR Problem

As mentioned in section 3.3, we use connectionist systems to represent the various functions associated with our algorithm. The issue of function approximation in RL is a very contentious one. Most RL algorithms assume that a exact representation of the value and policy functions are available and have been shown to converge only under such assumptions. The effect of function approximation errors on the performance of RL algorithms has not been investigated properly and needs more study. (See discussion in section 2.6.) If we tie up the issue of function approximation with the testing of our algorithm, (for e.g. use a powerful universal function approximator such as a Multi-Layer Peceptron with a lot of hidden neurons, for representing the $\hat{Q}$ and $\hat{\pi}$ functions) and the resulting combination does not work well, we would not know which is the cause. So we carefully chose the form of the function approximator based on the form of the Q function and the policy π, of the test problem. To make this choice we need the following lemma.

Lemma: Consider a quadratic function $f(y) = y^T K y$, where K is a symmetric, positive semi-definite $n \times n$ matrix and $y \in R^n$ This function can also represented as follows:

$$f(y) = y^T K y = \sum_{i=1}^{n} f_i^2(y) \tag{3.5.2}$$

where, $f_i(y)$ is a linear function of y.

Proof: Since K is symmetric we can write it as:

$$K = S \Lambda S^T$$

where S is an orthogonal matrix and Λ is a diagonal matrix, with the eigen values of K along the diagonal. Since K is positive semi-definite we can write:

$$\begin{aligned} K &= S\Lambda^{\frac{1}{2}}\Lambda^{\frac{1}{2}}S^T \\ &= (S\Lambda^{\frac{1}{2}})(S\Lambda^{\frac{1}{2}})^T \\ &= RR^T \end{aligned}$$

where, $R = S\Lambda^{\frac{1}{2}}$ and $\Lambda^{\frac{1}{2}}$ is a diagonal matrix with the square roots of the eigen values of K along its diagonal. We can now write the function $f(y)$ as:

$$f(y) = y^T K y = y^T R R^T y = \|R^T y\|^2$$ □

As can be seen from (3.5.1) the $-Q^\star$ function in this case is a positive semi-definite quadratic function and hence can be represented as the sum of squares of $n+m$ linear functions. This is the representation we choose in this work. The $\hat{Q}$ network consists of two layers as shown in figure 3.5.1. The first layer consists of $n+m$ fully connected linear neurons. The second layer neuron takes the output of the first layer neurons, squares and adds them and negates the sum to produce the output. We need to negate the output since we are posing this as a maximization problem.

The policy in this case is defined as a linear function of the state. Hence we can use a single layer of m linear neurons for the $\hat{\pi}$ network, as shown in figure 3.5.1. The output of the policy network forms a part of the input to the Q network, while the present state is fed as input to both the networks.

3.5.3 Numerical Results

In this section we present simulation results for the chosen test problem. We demonstrate the convergence of our algorithm on the LQR problem associated with the *double integrator* for ease of comparison with analytical results. For the double integrator in two dimensions, the matrices involved in the problem definition are:

$$n = 2 \; ; \; m = 1$$

$$A = \begin{pmatrix} 1 & T \\ 0 & 1 \end{pmatrix} \; ; \; B = \begin{pmatrix} T^2/2 \\ T \end{pmatrix} \; ; \; E = \begin{pmatrix} \alpha & 0 \\ 0 & \beta \end{pmatrix} \; ; \; F = 1$$

where T, α and β ($\in R$) are system parameters.
For our simulations we chose arbitrarily the following values:

$$T = 0.7 \; ; \; \alpha = 0.1 \; ; \; \beta = 0.5 \; ; \; \gamma = 1.0$$

For these values the optimal policy is given by:

$$U^\star = \begin{pmatrix} -0.219 \\ -0.823 \end{pmatrix}$$

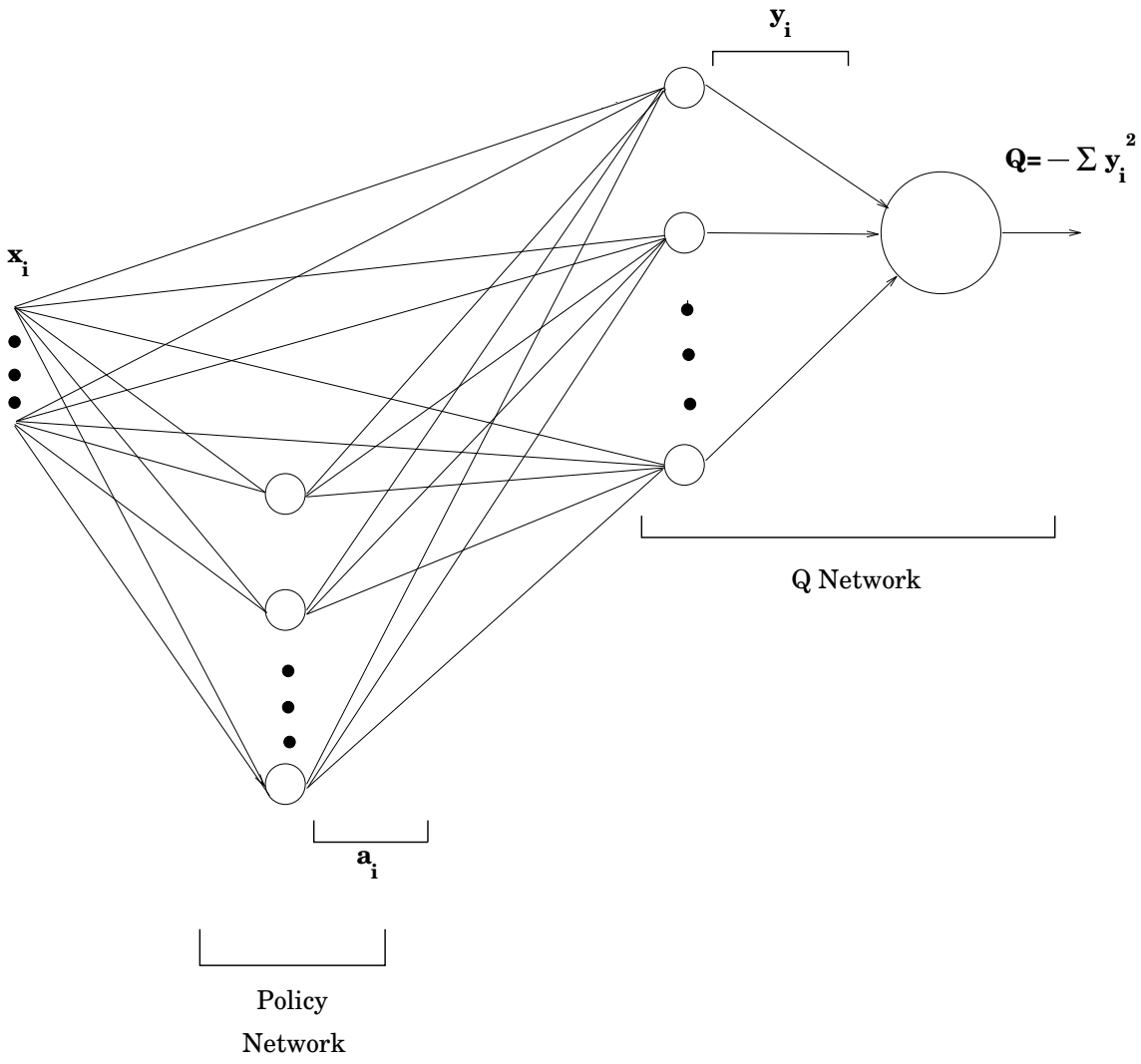

Figure 3.5.1 Connectionist network for the LQR problem

and the $Q^\star$ function is given by:

$$Q^\star(x,a) = -\begin{pmatrix} x^T & a \end{pmatrix} \begin{pmatrix} 0.638 & 0.843 & 0.458 \\ 0.843 & 3.255 & 1.722 \\ 0.458 & 1.722 & 2.093 \end{pmatrix} \begin{pmatrix} x \\ a \end{pmatrix}$$

The Q network now has 3 linear neurons in the first layer and 1 output neuron of the type described in the previous section. The policy network just consists of 1 linear neuron whose output is fed to the Q network.

When expressed as an actual update rule used in a connectionist network, (3.3.2) becomes:

$$v := v + \beta e \frac{\partial \hat{Q}(x,a)}{\partial v}|_{x=x_t;a=a_t} \tag{3.5.3}$$

where,

$$e = \left[r(x_t,a_t) + \gamma \hat{Q}(x_{t+1}, \hat{\pi}(x_{t+1}); v) - \hat{Q}(x_t,a_t;v) \right]$$

We decided to adopt the gradient ascent scheme outlined in section 3.3 since the problem has a Q function that is unimodal in a for a given x. Initially we used the update rule given in (3.5.3) for the Q network and a corresponding connectionist update rule to that given in (3.3.3) for the policy network. We found that even if we fix the policy network and allow only the $\hat{Q}$ network to train the convergence to Q^π was very slow.

In the learning rule (3.5.3) we assume that while we update the network at x_t, the $\hat{Q}$ values at x_{t+1} do not change. In a connectionist implementation this is obviously not true and therefore in effect we are shooting at a moving target. We can overcome this problem if we try to minimize:

$$\left(\hat{Q}(x_t,a_t) - (r(x_t,a_t) + \gamma \hat{Q}(x_{t+1}, \hat{\pi}(x_{t+1}))) \right)^2$$

instead. This is the idea behind the residual gradient approach recommended by Baird and Harmon (1993) and Werbos (1987). The modified learning rule is:

$$v := v - \beta e \left(\gamma \frac{\partial \hat{Q}(x,a)}{\partial v}|_{x=x_{t+1};a=a_{t+1}} - \frac{\partial \hat{Q}(x,a)}{\partial v}|_{x=x_t;a=a_t} \right) \tag{3.5.4}$$

Using the modified rule, the $\hat{Q}$ network converged to Q^π of a fixed policy about 10 times faster than before.

While performing the simulations we found that the policy network had to be updated at a much slower pace than the Q network. We chose learning rates in the ratio of $1 : 10^{-3}$ for the Q and policy networks respectively. We also found it advantageous to update the policy network only once for every few updates of the Q network.

Choosing the initial values for the policy and Q networks is an important issue. If the initial policy chosen is not a stabilizing one, the system tends to become unstable and learning is impossible. So we chose an arbitrary stabilizing policy as the initial one. Similar initializing policies have been shown by Bradtke (1993) to be essential for the convergence of his policy iteration scheme based on Q functions. Also we chose the initial Q as:

$$Q^\star(x,a) = -\begin{pmatrix} x^T & a \end{pmatrix} \begin{pmatrix} E & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ a \end{pmatrix}$$

This amounts to setting $Q = r$. This was done just for the sake of expediting learning. Over many runs we observed that the learning system converged to the optimal functions. Typical values of $\hat{Q}$ and $\hat{\pi}$ networks after a sufficient number of iterations were:

$$\hat{Q}(x,a) = -\begin{pmatrix} x^T & a \end{pmatrix} \begin{pmatrix} 0.621 & 0.813 & 0.440 \\ 0.813 & 3.237 & 1.721 \\ 0.440 & 1.721 & 2.036 \end{pmatrix} \begin{pmatrix} x \\ a \end{pmatrix}$$

$$\hat{\pi}(x) = \begin{pmatrix} -0.226 \\ -0.820 \end{pmatrix} x$$

We then decided to try out the general scheme outlined in section 3.3. In this problem finding the action corresponding to the maximum Q turns out to be particularly easy. A closed form expression can be derived for the max. action in terms of the weights of the Q network and the state inputs. We found that provided the network is trained sufficiently slowly it converges to the optimal values. In the simulations we conducted we could not see any significant advantage of using one scheme over the other. This is clearly because the gradient ascent scheme is particularly good for this problem.

3.6 Conclusion

In this chapter we discussed the difficulties of solving delayed RL problems having continuous action spaces. We briefly presented previous work done in tackling this problem and pointed out their drawbacks. We then proposed a extension of Q–learning applicable to continuous action spaces. This method overcomes the drawbacks associated with the earlier attempts. We also proposed a gradient ascent scheme applicable in the case of certain problems where the Q function is unimodal in a for a fixed x.

Chapter 4

Conclusion

Majority of the optimal control problems arising in engineering applications have continuous action spaces. However very little work has been done to design efficient delayed RL algorithms for such problems. In this thesis we have developed some useful initial ideas for this design. We have devised a simple scheme for extending Q–learning to continuous action spaces by augmenting the Q network with a policy network that adapts a policy that is optimal with respect to th Q function. For special, yet useful, class of Linear Regulation problems having concave reward (convex cost) functions we have shown that a simple gradient ascent update rule can be used for the policy network. We have demonstrated the working of our method by simulating it on the Linear Quadratic Regulation problem.

The thesis has provided only some initial ideas for the extension of Q–learning to continuous action spaces. Much more work is needed to establish its real usefulness. First, a general implementation of our method by employing universal function approximation techniques such as Multi-Layer Perceptrons or Radial Basis Function networks and demonstrating its working on non-trivial applications has to be carried out. Here, it will be interesting to consider the use of ontogenic networks such as the Resource Allocation Network. Second, it is important to theoretically investigate issues of convergence of the various algorithms suggested for continuous action spaces. The only result available thus far is that of Bradtke (1993), who has considered the special case of the Linear Quadratic Regulation problem and has shown that if his algorithm is started from a stabilizing policy then convegence occurs. It appears that proving convergence of the other general algorithms is a very hard

and challenging work. Another direction for research is to extend our ideas to continuous time operation. One way of doing this would be to use the advantage function (Baird 1993) instead of the Q function and suitably modify the learning rules. We hope to take up some of these problems in the future.

Bibliography

[1] 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.

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

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

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

[5] 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.

[6] 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.

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

[8] L. C. Baird III, 1993, Advantage updating. Wright-Patterson Air Force Base, Ohio, USA (Wright Laboratory Technical Report WL-TR-93-1146, available from the De-

fence Technical Information Center, Cameron Station, Alexandria, VA 22304-6145, USA.)

[9] L. C. Baird III and M. E. Harmon, In Preparation, Residual gradient algorithms. Technical Report, Wright-Patterson Air Force Base, Ohio, USA.

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

[11] A. G. Barto, 1986, Game–theoretic 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.

[12] A. G. Barto, 1992, Reinforcement 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] H. R. Berenji and P. Khedkar, 1992, Learning and tuning fuzzy logic controllers through reinforcements. IEEE Transactions on Neural Networks, 3:724–740.

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

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

[27] D. P. Bertsekas, 1994, A counter example to temporal-differences learning. Neural Computation, Vol. 7, No. 2.

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

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

[30] J. A. Boyen and A. W. Moore, 1995, Generalization in reinforement learning: Safely approximating the value function, In *Advances in Neural Information Processing Systems 7*, G. Tesauro, D. S. Touretzky and T. K. Leen, editors, pages 369–376, MIT Press, Cambridge, MA, USA.

[31] 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.

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

[33] S. J. Bradtke and M. O. Duff, 1995, Reinforcement learning methods for continuous–time Markov decision problems. In *Advances in Neural Information Processing Systems 7*, G.Tesauro, D.Touretzky, and T.Leen, editors, MIT Press, Cambridge, MA, USA.

[34] 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.

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

[36] 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.

[37] D. Chapman, 1991, *Vision, Instruction, and Action*. MIT Press, MA, USA.

[38] 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.*

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

[40] P. Cichosz, 1994, Reinforcement learning algorithms based on the methods of temporal differences, Masters Thesis, Institute of Computer Science, Warsaw University of Technology, Warsaw, Poland.

[41] P. Cichosz, 1995, Truncating temporal differences: On the efficient implementation of TD(λ) for reinforcement learning. Journal of Artificial Intelligence Research, 2:287–318.

[42] R. H. Crites and A. G. Barto, 1996, Improving elevator performance using reinforcement learning. To appear in *Advances in Neural Information Processing Systems 8*, MIT press, 1996.

[43] 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.

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

[45] P. Dayan, 1993, Improving generalization for temporal difference learning: The successor representation. Neural Computation, 5:613–624.

[46] 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.

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

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

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

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

[51] 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.

[52] 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.

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

[54] M. E. Harmon and L. C. Baird, III, and A. H. Klopf, 1995, Advantage updating applied to a differential game. In *Advances in Neural Information Processing Systems 7*, G.Tesauro, D.Touretzky, and T.Leen, editors, MIT Press, Cambridge, MA, USA.

[55] S. Haykin, 1994, Neural Networks: A Comprehensive foundation. Macmillan, NewYork, NY.

[56] M. Heger, 1994, Consideration of risk in reinforcement learning. In Proc. of 11th International Machine Learning Conference, ML-94.

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

[58] 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.

[59] T. Jaakkola, S. P. Singh, and M. I. Jordan, 1995, Reinforcement learning algorithm for partially observable Markov decision processes. In *Advances in Neural Information Processing Systems 7*, G.Tesauro, D.Touretzky, and T.Leen, editors, MIT Press, Cambridge, MA, USA.

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

[61] 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.

[62] 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.

[63] 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.

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

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

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

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

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

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

[70] 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.

[71] 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.

[72] 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.

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

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

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

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

[77] 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.

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

[79] 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.

[80] 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.

[81] M. A. F. McDonald and P. Hingston, 1994, Approximate discounted dynamic programming is unreliable. Technical Report 94/6, Department of Computer Science, The University of Western Australia, Crawley, WA, 6009.

[82] 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.

[83] 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.

[84] 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.

[85] 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.)

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

[87] 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.

[88] A. W. Moore, 1994, The parti-game algorithm for variable resolution reinforcement learning in multi-dimensional spaces. In *Advances in Neural Information Processing Systems 6*, J. D. Cowan, G. Tesauro, and J. Alspector, editors, pages Morgan Kaufmann, San Fransisco, CA, USA.

[89] 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.

[90] 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.

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

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

[93] 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.

[94] V. V. Phansalkar and M. A. L. Thathachar, 1995, Local and global optimization algorithms for generalized learning automata. *Neural Computation*, 7:950–973.

[95] 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.

[96] 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.

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

[98] 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.

[99] 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.)

[100] 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.

[101] N. N. Schraudolph, P. Dayan, and T. J. Sejnowski, 1994, Temporal difference learning of position evaluation in the game of Go. In *Advances in Neural Information Processing Systems 6*, J. D. Cowan, G. Tesauro, and J. Alspector, editors, pages Morgan Kaufmann, San Fransisco, CA, USA.

[102] 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.

[103] G. Shantaram, P. S. Shastry and M. A. L. Thathachar, 1994, Continuous action set learning automata for stochastic optimization. *Journal of the Franklin Institute*, Vol. 331B, No. 5, pages 607–628.

[104] 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.

[105] 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.

[106] 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.

[107] 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.

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

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

[110] S. P. Singh, 1994, Learning to solve Markovian decision processes, Ph.D Thesis, Department of Computer Science, University of Massachussetts, Amherst, MA, USA.

[111] 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.

[112] S. P. Singh, T. Jaakkola, and M. I. Jordan, 1994, Learning without state–estimation in partially observable Markov decision processes. Submitted to *Machine Learning*.

[113] 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.

[114] D. A. Sofge and D. A. White, 1990, Neural network based process optimization and control. In *Proceedings of the* 29^{th} *IEEE Conference on Decision and Control*, Honolulu, Hawaii, USA.

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

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

[117] 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.

[118] 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.

[119] 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.

[120] R. S. Sutton, 1996, Genreralization in reinforcement learning: successful examples using sparse coarse encoding. To appear in *Advances in Neural Information Processing Systems 8*, MIT press, 1996.

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

[122] 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.

[123] 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.

[124] 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.

[125] 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.

[126] 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.

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

[128] G. J. Tesauro, 1995, Temporal Difference Learning and TD-Gammon. *Communications of the ACM*, vol. 38, no. 3, pages 58–68.

[129] C. K. Tham and R. W. Prager, 1992, Reinforcement learning for multi-linked manipulator control, Technical Report CUED/F-INFENG/TR 104, Cambridge University Engineering Department, Cambridge CB2 1PZ, UK.

[130] 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.)

[131] M. A. L. Thathachar and V. V. Phansalkar, 1995, Convergence of teams and hierarchies of learning automata in connectionist systems. *IEEE Transactions on Systems, Man and Cybernetics*, vol. 25, no. 11, pages 1459–1469.

[132] 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.

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

[134] 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.

[135] 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.

[136] 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.

[137] J. N. Tsitsiklis and B. Van Roy, 1994, Feature-based methods for large scale dynamic programming. Technical Report, LIDS-P-2277, Laboratory for Information and Decision Systems, Massachussetts Institute of Technology, Cambridge, MA 02139, USA.

[138] 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.

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

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

[141] 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.*

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

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

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

[145] 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.

[146] 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.

[147] 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.

[148] 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.

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

[150] 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.

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

[152] R. J. Williams and L. C. Baird III, 1993a, Analysis of some incremental variants of policy iteration: First steps toward understanding actor–critic learning systems. Technical Report NU-CCS-93-11, College of Computer Science, Northeastern University, Boston, MA, USA.

[153] R. J. Williams and L. C. Baird III, 1993b, Tight performance bounds on greedy policies based on imperfect value functions. Technical Report NU-CCS-93-14, College of Computer Science, Northeastern University, Boston, MA, USA.

[154] W. Zhang and T. G. Dietterich, 1994, A reinforcement learning approach to job–shop scheduling. Technical Report, Computer Science Department, Oregon State University, Corvallis, Oregon, USA.