V Escola de Redes Neurais, Promoção: Conselho Nacional de Redes Neurais
pp. c028-c072, 19 de julho, 1999 - ITA, São José dos Campos - SP

Aprendizado por Reforço

Carlos Henrique Costa Ribeiro

Divisão de Ciência da Computação
Departamento de Teoria da Computação
Instituto Tecnológico de Aeronáutica
São José dos Campos, Brasil

Texto publicado originalmente (com ligeiras alterações) em CD-ROM na *International Joint Conference on Neural Networks (IJCNN'99) - Tutorial Track on Learning*, com o título *A Tutorial on Reinforcement Learning Techniques*.

Sumário

Aprendizado por Reforço (AR) é aprendizado via experimentação direta. Não assume a existência de um professor provedor de exemplos a partir do qual o aprendizado se desenvolve. Em AR, a experiência é a única professora. Com raizes históricas no estudo de reflexos condicionados, AR logo atraiu o interesse de Engenheiros e Cientistas da Computação por sua relevância teórica e aplicações potenciais em campos tão diversos quanto Pesquisa Operacional e Robótica.

Computacionalmente, AR opera em um ambiente de aprendizagem composto por dois elementos: o aprendiz e um processo dinâmico. A passos de tempo sucessivos, o aprendiz faz uma observação do estado de processo, seleciona uma ação e a aplica de volta ao processo. Sua meta é descobrir uma política de ações que controle o comportamento deste processo dinâmico, usando para isso sinais (reforços) indicativos de quão bem está sendo executada a tarefa em questão. Estes sinais normalmente são associados a alguma condição dramática— por exemplo, realização de uma subtarefa (recompensa) ou completo fracasso (castigo), e a meta do aprendiz é aperfeiçoar seu comportamento baseado em uma medida de desempenho (função dos reforços recebidos). O ponto crucial é que, para fazer isto, o aprendiz tem que avaliar as condições (associações entre estados observados e ações escolhidas) que produzem recompensas ou castigos. Em outras palavras, tem que aprender a *atribuir crédito* a ações e estados passados, através de uma correta estimação dos custos associados a estes eventos.

A partir de conceitos básicos, este texto apresenta os vários tipos de algoritmos de Aprendizado por Reforço, desenvolve as ferramentas matemáticas correspondentes, avalia suas limitações práticas e discute alternativas que têm sido propostas para aplicar AR a tarefas realistas, como aquelas envolvendo grandes espaços de estados ou observabilidade parcial. Exemplos e diagramas ilustram os pontos principais, e muitas referências para a literatura especializada e para *sites* da Internet onde podem ser obtidas demonstrações e informação adicional são indicados.

Abstract

Reinforcement Learning (RL) is learning through direct experimentation. It does not assume the existence of a teacher that provides examples upon which learning of a task takes place. Instead, in RL experience is the only teacher. With historical roots on the study of conditioned reflexes, RL soon attracted the interest of Engineers and Computer Scientists because of its theoretical relevance and potential applications in fields as diverse as Operational Research and Robotics.

Computationally, RL is intended to operate in a learning environment composed by two subjects: the learner and a dynamic process. At successive time steps, the learner makes an observation of the process state, selects an action and applies it back to the process. The goal of the learner is to find out an action policy that controls the behavior of this dynamic process, guided by signals (reinforcements) that indicate how well it is performing the required task. These signals are usually associated to some dramatic condition — e.g., accomplishment of a subtask (reward) or complete failure (punishment), and the learner's goal is to optimize its behavior based on some performance measure (a function of the received reinforcements). The crucial point is that in order to do that, the learner must evaluate the conditions (associations between observed states and chosen actions) that lead to rewards or punishments. In other words, it must learn how to assign credit to past actions and states by correctly estimating costs associated to these events.

Starting from basic concepts, this tutorial presents the many flavors of RL algorithms, develops the corresponding mathematical tools, assess their practical limitations and discusses alternatives that have been proposed for applying RL to realistic tasks, such as those involving large state spaces or partial observability. It relies on examples and diagrams to illustrate the main points, and provides many references to the specialized literature and to Internet sites where relevant demos and additional information can be obtained.

Contents

1 Introduction c033
- 1.1 Reinforcement Learning Agents . . . c033
- 1.2 Generalization and Approximation in RL . . . c035
- 1.3 Perceptual Limitations: The Achilles Heel of RL . . . c035
- 1.4 Summary of Chapters . . . c036

2 Reinforcement Learning: The Basics c037
- 2.1 What Kind of Learning Agent? . . . c037
- 2.2 A General Model . . . c039
- 2.3 An Example: The Cartpole Problem . . . c040
- 2.4 Markov Decision Processes . . . c041
  - 2.4.1 The Markovian Condition . . . c041
  - 2.4.2 Solving MDP Problems: Dynamic Programming . . . c042
- 2.5 From DP to RL: Monte Carlo Simulation . . . c044
- 2.6 Reinforcement Learning . . . c044
  - 2.6.1 The Temporal Differences Method . . . c044
  - 2.6.2 Q-Learning . . . c046
  - 2.6.3 Why is Q-Learning so Popular? . . . c047
  - 2.6.4 Alternatives to Q-Learning . . . c048
  - 2.6.5 Other RL techniques: A Brief History . . . c049

3 Experience Generalization c052
- 3.1 The Exploration/Exploitation Tradeoff . . . c052
- 3.2 Accelerating Exploration . . . c052
- 3.3 Experience Generalization . . . c053
  - 3.3.1 Experience Generalization in Lookup Tables . . . c053
  - 3.3.2 Experience Generalization from Function Approximation . . . c055
- 3.4 Iterative Clustering (Splitting) . . . c060

4 Partially Observable Processes c062
- 4.1 Partial Observability and DP . . . c062
- 4.2 Partial Observability and RL . . . c063
  - 4.2.1 Attention-Based Methods . . . c063
  - 4.2.2 Memory-based Methods . . . c064
  - 4.2.3 Combining Attention and Memory . . . c065

A Internet Sites on RL-related subjects c067

List of Figures

1.1 Interaction between agent and dynamic process in Reinforcement Learning. The agent observes the states of the system through its sensors and chooses actions based on cost estimates which encode its cumulated experience. The only available performance signals are the reinforcements (rewards and punishments) provided by the process. c034

2.1 A general model of the learning agent. c039
2.2 The cartpole system. c040
2.3 Look-up table implementation in Q-learning. Each table separately stores all the action values for each action. c048

3.1 Structural and temporal credit assignment. The dark circles are states visited in a temporal sequence $t, t+1, t+2, t+3, t+4$. The curved arrows represent propagation of temporal updates, and the dashed arrows represent propagation of the structural credit assignment among similar states that are not visited in sequence. c054
3.2 Feature-based compact representation model. c055
3.3 A two-states Markov chain for which a least-squares approximation method for cost calculation may diverge. c056
3.4 An Adaptive Heuristic Critic (AHC) agent. The Policy Module generates improved actions for a given policy using the costs estimated by the Evaluation Module. c058
3.5 The CMAC approach. Each tiling (in this case, T_1, T_2 and T_3) has a single tile associated with the state of the process. The hash function H then maps the tiles to different positions in U. c060

4.1 Memory-based methods. Case (a): use of feedback mechanism. Case (b): use of explicit memory. c065

Chapter 1

Introduction

This tutorial is about learning as it is usually understood by a layman: a method for knowledge acquisition through a voluntary autonomous procedure, via direct interaction with a dynamic system. Learning as an *active process*.

In studying learning as such, one of our goals will be to formalize the concept so that it can be used efficiently as an Engineering tool. We must define precisely what we mean by a learner and by an active learning process. It turns out that this area of study is now relatively well developed, although there are many open issues that we will try to address.

1.1 Reinforcement Learning Agents

The learning environment we will consider is a system composed by two subjects: the learning *agent* (or simply the *learner*) and a dynamic *process*. At successive time steps, the agent makes an *observation* of the process state, selects an action and applies it back to the process, modifying the state. The goal of the agent is to find out adequate actions for controlling this process. In order to do that in an *autonomous* way, it uses a technique known as *Reinforcement Learning*.

Reinforcement Learning (RL for short) is learning through direct experimentation. It does not assume the existence of a teacher that provides 'training examples'. Instead, *in RL experience is the only teacher*. The learner acts on the process to receive signals (reinforcements) from it, indications about how well it is performing the required task. These signals are usually associated to some dramatic condition — *e.g.*, accomplishment of a subtask (reward) or complete failure (punishment), and the learner's goal is to optimize its behavior based on some performance measure (usually minimization of a *cost function*[1]). The crucial point is that in order to do that, in the RL framework the learning agent must *learn* the conditions (associations between observed states and chosen actions) that lead to rewards or punishments. In other words, it must learn how to *assign credit* to past actions and states by correctly estimating costs associated to these events. This is in contrast with supervised learning (Haykin, 1999), where the credits are implicitly given beforehand as part of the training procedure. *RL agents are thus characterized by their autonomy.*

[1] Unless otherwise stated, minimization of a cost function is used throughout this text. Naturally, maximization of a reward function could be similarly used.

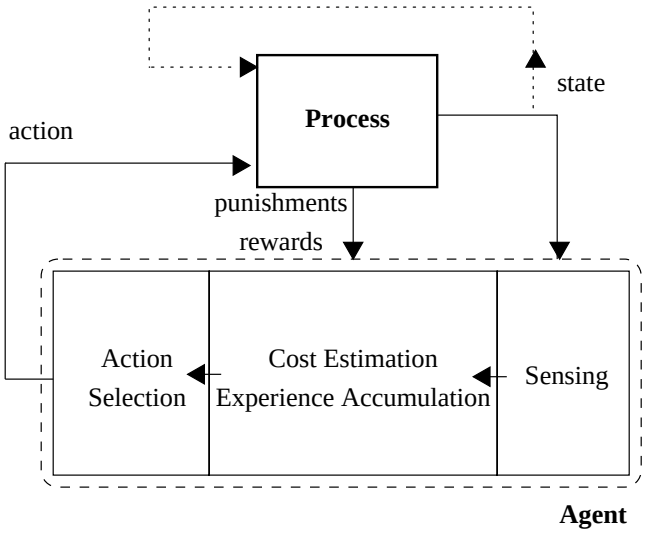

Figure 1.1: Interaction between agent and dynamic process in Reinforcement Learning. The agent observes the states of the system through its sensors and chooses actions based on cost estimates which encode its cumulated experience. The only available performance signals are the reinforcements (rewards and punishments) provided by the process.

Figure 1.1 shows how the RL agent interacts with the process. Section 2.2 presents a slightly more realistic model for this interaction.

Historically, the earliest use of a RL method was in Samuel's Checkers player program (Samuel, 1959). After that, a long period of sporadic but insightful research (mainly based on attempts at creating computational models for conditioned reflex phenomena (Sutton and Barto, 1990)) was interrupted by a few theoretical (Sutton, 1988) and practical (Lin, 1992; Tesauro, 1992; Barto et al., 1983) developments of autonomous learning models. Research in RL as a Computing technique then skyrocketed when its important relationship with Optimal Control got initiated (Barto et al., 1990) and further developed by Singh and others (Singh, 1994; Jaakkola et al., 1994). Indeed, terms from the Optimal Control jargon such as 'optimization' and 'performance measure' are also part of the basic RL vocabulary.

Nearly all RL methods currently in use are based on the Temporal Differences (TD) technique (Sutton, 1988). The fundamental idea behind it is *prediction learning*: when the agent receives a reinforcement, it must somehow propagate it backwards in time so that states leading to that condition and formerly visited may be associated with a prediction of future consequences. This is based on an important assumption on the process' dynamics, called the *Markov condition*: the present observation must be perfectly a conditional probability on the immediate past observation and input action. In practical terms, this means that the agent's sensors must be good enough to produce correct and unambiguous observations of the process states.

1.2 Generalization and Approximation in RL

Getting learning information through direct experimentation is a practice that is usually not under complete control by the agent: it can choose actions but cannot determine in advance the consequences of these actions for every state because it usually does not have a sufficiently precise model of the process upon which to base judgments. Instead, it must estimate the expected costs through direct state visitation; it must choose an action, observe the result and propagate it backwards in time, following the TD procedure mentioned above.

Correct determination of action consequences depends on a reasonable number of updates carried out for each state visited. Actually, convergence of stochastic adaptive algorithms in general[2] is conditional on an infinite number of visits to every possible process state. The impossibility of fulfilling this condition creates the exploration/exploitation tradeoff: the agent must try alternative action policies which allows a better exploration (and consequently a better model) of the state space, but at the same time it must consider that good performance is attained only if the best action policy is exploited. In the Control literature, this conflict between the parameter estimation objective (determining a correct model of the process) and the control objective (exercising the best possible action policy) is referred to as the *dual control problem* (Bertsekas, 1995a).

Direct state visitation as the only means of acquiring knowledge causes the problem above. The agent should be able to *generalize* the costly experience it gains from visiting a state. In fact, in the last few years experience generalization has been acknowledged as an extremely important issue to be tackled if RL methods are to be considered useful engineering tools (Tesauro, 1992).

This problem of generalization is closely related to function approximation: the capacity of representing an input/output relationship by using far fewer parameters than classification features. Approximations are worth studying because of its practical relevance (very few real problems allow for a solution based on explicit state and action listings), and also because they have generalization as an *emergent property*.

1.3 Perceptual Limitations: The Achilles Heel of RL

We mentioned that the Markov condition is a necessary assumption for application of the Temporal Differences technique. Unfortunately, learning agents are frequently subject to perceptual limitations that keep them from completely discriminating process states. In RL this cause the *perceptual aliasing* problem: different states may have the same representation for the agent, and as a consequence the same expected costs will be assigned to them, even when this should not be the case. In practice, there are two possible approaches for this problem (Lin and Mitchell, 1992): either using past observations as an additional source of information about the present state, or considering *attentional* agents that can actively select observations. The perceptual aliasing problem, however, is a largely open issue, both from the theoretical and practical points of view.

[2] RL algorithms are particular forms of stochastic adaptation.

1.4 Summary of Chapters

Chapter 2 presents the basic RL model and reviews the literature on learning methods. It explains how an autonomous learning agent operates and why TD techniques are suitable, using as a starting point the theory of Markov Decision Processes and Dynamic Programming. The basic RL techniques are discussed assuming ideal conditions of observability and dimensionality.

Chapters 3 and 4 put RL in a realistic Engineering perspective. In chapter 3, the problem of function approximation and generalization is studied in the context of RL. Chapter 4 is about the perceptual aliasing problem, a condition found in many practical applications of autonomous learning.

Chapter 2

Reinforcement Learning: The Basics

We now define a general model for Reinforcement Learning, based on the concept of autonomy. Learning techniques for it will be analyzed, with an emphasis on the Temporal Differences approach and its relationship with Optimal Control. Later on, we will briefly discuss some variations of basic algorithms and other methods of historical significance.

2.1 What Kind of Learning Agent?

The term *self-improving reactive agent* was used by (Lin, 1991) to define the kind of learning agent that is of interest for us. A generic definition for autonomous agents, on the other hand, is given in (Russell and Norvig, 1995):

> An agent is autonomous to the extent that its behavior is determined by its own experience.

An agent is meant to be the entity that communicates with and tries to control an external process by taking appropriate actions. These actions are collectively called *action policy*[1]. A *reactive* agent tries to control the process by using an action policy mapped *directly* from its *observations* (instantly available information about the states of the process) and internal conditions. Any output from this mapping is meant to be derived from a simple *condition-action* rule. Once the agent selects an action and applies it, the process changes its state (as a function of the action taken, past state and some random disturbance) and presents it as a new observation to the agent. The agent then takes another immediate action based on this new observation and possibly on some internal condition[2].

[1] 'Behavior' is sometimes referred in the literature as a kind of high level control strategy (Matarić, 1990), that is, a subset of an action policy.

[2] The concept of reactiveness has been subject to some confusion. Our agents are reactive in a dynamic response sense: they map actions *directly* from observation and internal conditions, without explicit deliberation. An alternative definition says that a system is reactive if it minimizes the use of internal state (Gat et al., 1994). We are mainly concerned with autonomous learning for realistic dynamic processes, thus the emphasis on a dynamics-related property

What then characterizes autonomy? The key is to know what is meant by *own experience*. Experience itself is a concept closely related to the idea of learning or adaptation: it improves along time, and if behavior is a direct function of experience it must also improve. The use of experience as such is characteristic of learning techniques in general, and is exemplified by the standard *supervisory* methods for feedforward neural networks, where learning is guided by an external teacher that presents the experiments (training pairs).

Learning through the use of *own experience* — *i.e.*, self-improvement — is a stronger concept. It implies that the agent itself must generate its experience, without the help of a teacher. In this case, the only source of learning is the actual interaction between agent and process, which must nevertheless give some indications about overall performance. These indications (called *reinforcements*) are much less informative than the training examples used in supervisory methods[3].

Still, the definition given above is too stringent as it characterizes behavior as a function solely of experience. It is convenient to include *built-in* knowledge as an additional determinant of behavior. In fact, basic *a priori* decisions such as choice of learning algorithm, architecture and representational features should be considered as prior knowledge, because they are normally made available to the RL agent before it starts the learning process. But less obvious built-in knowledge may also be important: it has been argued (Brooks and Mataric, 1993) and demonstrated (del R. Millán, 1996; Kuipers, 1987) that realistic agents combining learning and rather elaborated built-in knowledge (represented for instance by a knowledge of action policy strategies for escaping from 'dangerous' situations) are usually much more successful than those purely based on *tabula rasa* learning. Notice that this also makes sense if one is interested in biological modeling, as evolution provides living beings with built-in behavioral determinants.

A more precise definition of a learning agent in the sense we consider here would then be[4]:

> An agent that develops through own experience and with the help of some built-in knowledge an action policy directly mapped from its observations and internal conditions.

The study of Reinforcement Learning agents used to have a strong biological motivation (Sutton and Barto, 1990), but in the last few years the enthusiasm switched towards the Engineering applications of the idea. The great majority of these Engineering applications, however, are directed towards problem solving in simulated situations, where a model is available and the cost of experimenting is simply translated in terms of processor time. In these cases, RL algorithms have been mainly used as alternatives to more traditional optimality techniques. Undoubtedly, many interesting issues emerged from this model-based RL framework (a comprehensive review of related results can be found in (Bertsekas and Tsitsiklis, 1996)). Nevertheless, for many realistic applications (*e.g*, robot learning), a suitable model may not be available and action policies must be learned with a minimal number of trials, due to the additional costs involved (Brooks and Mataric, 1993). These are the cases where experience is

[3]The reinforcements are normally associated to visitation to some particular states, and could then be considered as part of the agent's state observation. Pedagogically, however, it is convenient to make the distinction.

[4]Henceforth, the terms *learning agent*, *autonomous agent* and *Reinforcement Learning agent* will be used interchangeably to describe this kind of agent.

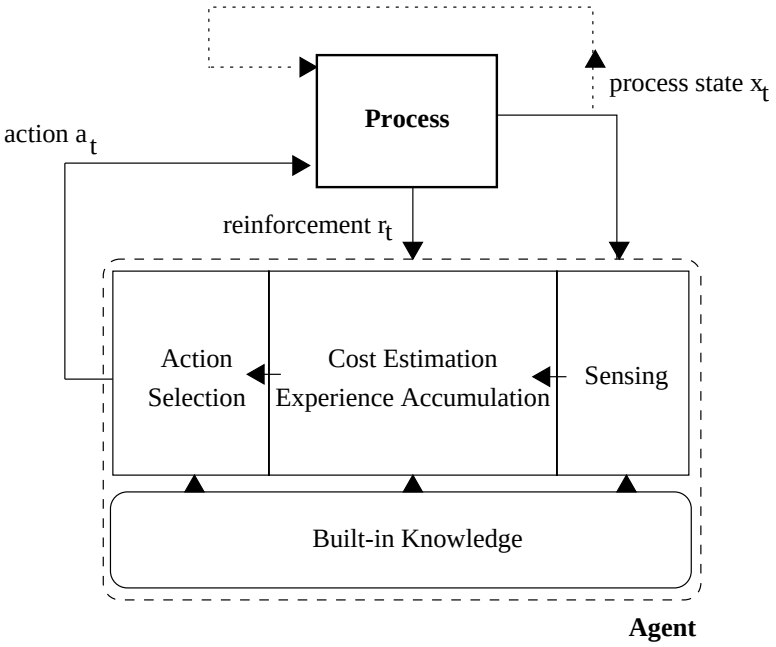

Figure 2.1: A general model of the learning agent.

expensive, and correspond to the fundamental problem faced by the learning agent defined above.

2.2 A General Model

We can now define a general model for our learning agent. Referring to figure 2.1, the accumulation of experience that guides the behavior (action policy) is represented by a *cost estimator* whose parameters are learned as new experiences are presented to the agent. The agent is also equipped with *sensors* that defines how *observations* about the external process are made. These observations may be — if necessary — combined with past observations or input to a state estimator, defining an *information vector* or *internal state* which represents the agent's belief about the real state of the process. The cost estimator then maps these internal states and presented reinforcements to associated costs, which are basically expectations about how good or bad these states are, given the experience obtained so far. Finally, these costs guide the action policy. The built-in knowledge may affect the behavior of the agent either directly, altering the action policy or indirectly, influencing the cost estimator or sensors.

The experience accumulation and action taking process is represented by the following sequence. At a certain instant of time, the agent:

1. Makes an observation and perceives any reinforcement signal provided by the process.

2. Takes an action based on the former experience associated with the current observation and reinforcement.

3. Makes a new observation and updates its cumulated experience.

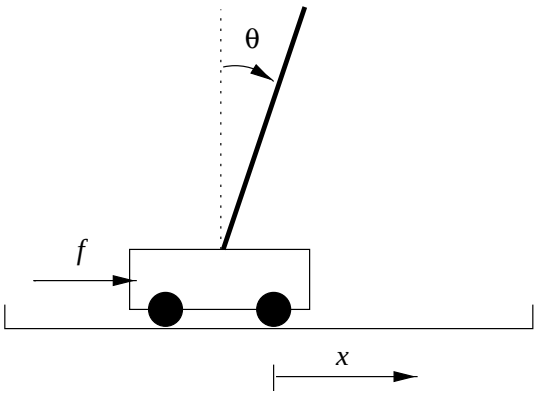

Figure 2.2: The cartpole system.

2.3 An Example: The Cartpole Problem

The so-called *cartpole problem* has been frequently used for demonstrations on the capabilities and limitations of self-improving agents (Barto et al., 1983; Michie and Chambers, 1968; Widrow and Smith, 1963). The problem consists in finding an appropriate action policy that balances a pole on the top of a cart for as long as possible, whilst at the same time keeping the cart between limiting track positions (figure 2.2). The actions are represented by a force applied horizontally to the cart, usually assuming one out of two possible values $-f$ or $+f$ (*bang-bang* control). The cartpole system itself is described by a set of four state variables $(x, \dot{x}, \theta, \dot{\theta})$, that represent respectively the position of the cart with respect to the center of the track, the linear velocity, the angle of the pole with respect to the vertical axis and the angular velocity. 'To balance' means to keep the cart between the limits of the track and to prevent the pole from falling down. Although designing a standard controller for the cartpole system from its state equations is a simple textbook task (see for instance (Ogata, 1994)), the problem is more complicated when there is no model available. In the RL context, the only reinforcement signal usually considered is a negative-valued cost for state transitions that lead to a failure condition (cart hitting the track limits or pole falling down). Learning through direct experimentation in this case means that the agent must try to control the system a number of times but always receiving a reinforcement only when it fails to do so, until a satisfactory action policy is found.

In this case, the experience accumulation and action taking process described above is as follows. At time t, the agent observes the process state $(x_t, \dot{x}_t, \theta_t, \dot{\theta}_t)$ (or a partial observation of it). It then takes an action, observes the resulting state and receives the reinforcement signal, which is always zero unless a failure condition occurs. Finally, it updates the values stored in the cost estimator by recording the immediate outcome (failure or success) of this new experience.

The cartpole problem has historical importance for Reinforcement Learning: it was one of the first successful applications of an algorithm based on model-free action policy estimation, as described in 1983 by Andrew Barto and Richard Sutton in their pioneering paper (Barto et al., 1983). The source code for the original algorithm (which is based on the AHC method described later on in this chapter) can still be found at the URL site

- ftp://ftp.cs.umass.edu/pub/anw/pub/sutton/pole.c

It is well worth downloading and running it: starting from no knowledge whatsoever of the system dynamics, the algorithm manages to find an adequate action policy in very short time.

2.4 Markov Decision Processes

The study of RL agents is greatly facilitated if a convenient mathematical formalism is adopted. This formalism — known as Markov Decision Processes (MDPs) — is well established, but assumes a simplifying condition on the agent which is, however, largely compensated by the gained ease of analysis.

2.4.1 The Markovian Condition

The basic assumption on the study of Markov Decision Processes is the *Markov condition*: any observation $\mathbf{o}$ made by the agent must be a function only of its last observation and action (plus some random disturbance):

$$\mathbf{o}_{t+1} = f(\mathbf{o}_t, a_t, w_t) \tag{2.1}$$

where $\mathbf{o}_t$ is the observation at time t, a_t is the action taken and w_t is the disturbance. Naturally, if the agent can faithfully observe the states of the process — which by definition summarize all the relevant information about the process dynamics at a certain instant of time — then its observations are Markovian. On the other hand, if the observations made by the agent are not sufficient to summarize *all* the information about the process, a non-Markovian condition takes place:

$$\mathbf{o}_{t+1} = f(\mathbf{o}_t, \mathbf{o}_{t-1}, \mathbf{o}_{t-2}, \ldots, a_t, a_{t-1}, a_{t-2}, \ldots, w_t) \tag{2.2}$$

In principle, it is possible to transform a non-Markovian problem into a Markovian one, through the use of *information vectors* that summarize all the *available* information about the process (Striebel, 1965). In general, however, the computational requirements for this are overwhelming, and the imperfect state information case permits only suboptimal solutions (Bertsekas, 1995b).

In this chapter we analyze only the case in which $\mathbf{o}_t$ provides complete information about $\mathbf{x}_t$[5]. This is equivalent to perfect observability of states, which guarantees the Markov condition and thus a chance to get the optimal solution of the so-called Markov Decision Process problem defined below.

The disturbance w_t introduces a stochastic element on the process behavior. It is usually more adequate to express the dynamic of the process through a collection of conditional transition probabilities $P(\mathbf{x}_{t+1}|\mathbf{x}_t, a_t)$.

Of particular interest is the discounted infinite horizon formulation of the Markov Decision Process problem. Given

- A finite set of possible actions $a \in \mathcal{A}$,
- A finite set of process states $\mathbf{x} \in \mathcal{X}$,
- A stationary discrete-time stochastic process, modeled by transition probabilities $P(\mathbf{x}_{t+1}|\mathbf{x}_t, a_t)$ and

[5]This means that we can indifferently use $\mathbf{x}_t$ or $\mathbf{o}_t$.

- A finite set of bounded reinforcements (payoffs) $r(\mathbf{x}, a) \in \mathbb{R}$;

The agent must try to find out a stationary policy of actions $a_t^* = \pi^*(\mathbf{x}_t)$ which minimizes the *expected cost function*:

$$V^{\pi}(\mathbf{x}_0) = \lim_{M \to \infty} \mathrm{E}[\sum_{t=0}^{M} \gamma^t r(\mathbf{x}_t, \pi(\mathbf{x}_t))] \tag{2.3}$$

for every state $\mathbf{x}_0$. The superscript π indicates the dependency on the followed action policy, via the transition probabilities $P(\mathbf{x}_{t+1}|\mathbf{x}_t, a_t = \pi(x_t))$. The *discount factor* $0 \leq \gamma < 1$ forces recent reinforcements to be more important than remote ones. It can be shown that the *optimal cost function*

$$V^*(\mathbf{x}_0) = \lim_{M \to \infty} \mathrm{E}[\sum_{t=0}^{M} \gamma^t r(\mathbf{x}_t, \pi^*(\mathbf{x}_t))] \tag{2.4}$$

is unique, although there can be more than a single optimal policy π^* (Puterman, 1994).

2.4.2 Solving MDP Problems: Dynamic Programming

Dynamic Programming (or DP for short) is a standard method that provides an optimal stationary policy π^* for the stochastic problem above. It actually encompasses a large collection of techniques, all of them based on a simple optimality principle and on some basic theorems.

Two useful operators that provide a shorthand notation for DP theorems are:

The Successive Approximation Operator T_{π} . For any cost function $V : \mathcal{X} \mapsto \mathbb{R}$ and action policy $\pi : \mathcal{X} \mapsto \mathcal{A}$,

$$(\mathrm{T}_{\pi} V)(\mathbf{x}) = r(\mathbf{x}, \pi(\mathbf{x})) + \gamma \sum_{\mathbf{y} \in \mathcal{X}} P(\mathbf{y}|\mathbf{x}, \pi(\mathbf{x})) V(\mathbf{y}) \tag{2.5}$$

The Value Iteration Operator T . For any cost function $V : \mathcal{X} \mapsto \mathbb{R}$,

$$(\mathrm{T}V)(\mathbf{x}) = \min_{a} [r(\mathbf{x}, a) + \gamma \sum_{\mathbf{y} \in \mathcal{X}} P(\mathbf{y}|\mathbf{x}, a) V(\mathbf{y})] \tag{2.6}$$

Basic Dynamic Programming Theorems

Below are presented the fundamental DP theorems for discounted infinite horizon problems. Proofs can be found in many textbooks, such as (Bellman, 1957; Bertsekas, 1995b; Puterman, 1994).

Theorem 1 *For any arbitrary bounded initial function $V(\mathbf{x})$ and state $\mathbf{x}$, the optimal cost function satisfies*

$$V^*(\mathbf{x}) = \lim_{N \to \infty} (\mathrm{T}^N V)(\mathbf{x}) \tag{2.7}$$

Theorem 2 *For every stationary action policy π and state $\mathbf{x}$, the cost function satisfies*

$$V_{\pi}(\mathbf{x}) = \lim_{N \to \infty} (\mathrm{T}_{\pi}^N V)(\mathbf{x}) \tag{2.8}$$

Theorem 3 (Bellman's Optimality Equation) *The optimal cost function V^* is the fixed point of the operator* T*, or in other words, V^* is the unique bounded function that satisfies*

$$V^* = \mathrm{T}V^* \tag{2.9}$$

Theorem 4 *For every stationary action policy π, the associated cost function V_π is the fixed point of the operator T_π, or in other words, V_π is the unique bounded function that satisfies*

$$V_\pi = \mathrm{T}_\pi V_\pi \tag{2.10}$$

Theorem 5 *A stationary action policy π is optimal if and only if*

$$\mathrm{T}_\pi V^* = \mathrm{T}V^* \tag{2.11}$$

Basic Dynamic Programming Techniques

There are two main classes of well-established methods for finding out optimal policies on MDPs. Both are based on the theorems above.

The Value Iteration method. This method consists simply on a recursive application of the operator T to an arbitrary initial approximation V of the optimal cost function. As V^* is the fixed point of T (Theorem 3), the method finds an optimal policy through

$$\begin{aligned} \pi^*(\mathbf{x}) &= \arg[\lim_{N\to\infty}(\mathrm{T}^N V)(\mathbf{x})] \\ &= \arg\min_a[r(\mathbf{x},a) + \gamma \sum_{\mathbf{y}\in\mathcal{X}} P(\mathbf{y}|\mathbf{x},a)V^*(\mathbf{y})] \end{aligned} \tag{2.12}$$

The Policy Iteration method. Given an initial policy π^0, two successive steps are performed in loop until π^* (or a good approximation) is found.

In the first step, the current policy π_k is evaluated (*i.e.*, has its associated cost calculated) either exactly, through the solution of the linear system of equations

$$V_{\pi_k} = \mathrm{T}_{\pi_k} V_{\pi_k} \tag{2.13}$$

or approximately, through a finite number M of applications of the Successive Approximation operator:

$$V_{\pi_k} \approx \mathrm{T}_{\pi_k}^M V_0 \tag{2.14}$$

This is the *Policy Evaluation* step.

In the second step, an improved stationary policy π^{k+1} is obtained through

$$\begin{aligned} \pi_{k+1}(\mathbf{x}) &= \arg[(\mathrm{T}V_{\pi_k})(\mathbf{x})] \\ &= \arg\min_a[r(\mathbf{x},a) + \gamma \sum_{\mathbf{y}\in\mathcal{X}} P(\mathbf{y}|\mathbf{x},a)V_{\pi_k}(\mathbf{y})] \end{aligned} \tag{2.15}$$

This policy π_{k+1} is called the *greedy* policy with respect to V_{π_k}.

2.5 From DP to RL: Monte Carlo Simulation

Dynamic Programming requires an explicit, complete model of the process to be controlled, represented by the availability of the transition probabilities. Autonomous learning, on the other hand, is completely based on interactive experience and does not require a model whatsoever. Before we get there, though, it is interesting to consider an intermediate case, which is a bridge linking DP and RL: the availability of a 'weak' model, which allows us to simulate generated sample transitions, without the need for a complete analytic description based on transition probabilities.

Let us start with the following problem: For a given fixed action policy π, how is it possible to calculate *by simulation — i.e.*, without the help of standard DP — the cost function V_π? The simplest solution is to run many simulated trajectories from each state and average the cumulative reinforcements obtained, doing a *Monte Carlo simulation*.

For a given state $\mathbf{x}_0$, let us denote $v_\pi(\mathbf{x}_0, m)$ the simulated discounted cumulative cost obtained after $\mathbf{x}_0$ is visited the mth time:

$$v_\pi(\mathbf{x}_0, m) = r(\mathbf{x}_0) + \gamma r(\mathbf{x}_1) + \gamma^2 r(\mathbf{x}_2) + \dots \tag{2.16}$$

where $\mathbf{x}_1, \mathbf{x}_2, \dots$ are the successively visited states for that particular run, and $r(\mathbf{x}_k) \equiv r(\mathbf{x}_k, \pi(\mathbf{x}_k))$, a simpler notation considering that the action policy is fixed. Assuming that the simulations correctly average the desired quantities, we have:

$$V_\pi(\mathbf{x}_0) = \mathrm{E}[v_\pi(\mathbf{x}_0, m)] = \lim_{M\to\infty} \frac{1}{M} \sum_{m=1}^{M} v_\pi(\mathbf{x}, m) \tag{2.17}$$

This average can be iteratively calculated by a Robbins-Monro procedure (Robbins and Monro, 1951):

$$V_\pi(\mathbf{x}_0) \leftarrow V_\pi(\mathbf{x}_0) + \alpha_m [v_\pi(\mathbf{x}_0, m) - V_\pi(\mathbf{x}_0)] \tag{2.18}$$

where $\alpha_m = 1/m$.

2.6 Reinforcement Learning

As mentioned in the last section, solving MDP problems through Dynamic Programming requires a model of the process to be controlled, as both the Successive Approximation and Value Iteration operators explicitly use the transition probabilities. Monte Carlo methods, on the other hand, use a weak simulation model that does not require calculation of transition probabilities, but does require complete runs before updates can be performed. Let us now analyse the basic RL algorithm, which allows for model-free *and* online updating as the learning process develops.

2.6.1 The Temporal Differences Method

Almost every RL technique currently in use is based on the Temporal Differences (TD) method, first presented by (Sutton, 1988). In contrast with previous attempts to implement the reinforcement idea, TD provides a very consistent mathematical framework.

Cost Evaluation by Temporal Differences

For on-line purposes, equation 2.18 suffers from a drawback: $V_\pi(\mathbf{x}_0)$ can only be updated after $v_\pi(\mathbf{x}_0, m)$ is calculated from a complete simulation run. The TD method provides an elegant solution to this problem by forcing updates immediately after visits to new states.

In order to derive Temporal Differences, let us first expand equation 2.18:

$$V_\pi(\mathbf{x}_0) \leftarrow V_\pi(\mathbf{x}_0) + \alpha_m[r(\mathbf{x}_0) + \gamma r(\mathbf{x}_1) + \gamma^2 r(\mathbf{x}_2) + \ldots - V_\pi(\mathbf{x}_0)] \tag{2.19}$$

Adding and subtracting terms, we get

$$\begin{aligned} V_\pi(\mathbf{x}_0) \leftarrow V_\pi(\mathbf{x}_0) + \alpha_m[\quad & (r(\mathbf{x}_0) + \gamma V_\pi(\mathbf{x}_1) - V_\pi(\mathbf{x}_0)) + \\ & \gamma(r(\mathbf{x}_1) + \gamma V_\pi(\mathbf{x}_2) - V_\pi(\mathbf{x}_1)) + \\ & \gamma^2(r(\mathbf{x}_2) + \gamma V_\pi(\mathbf{x}_3) - V_\pi(\mathbf{x}_2)) + \\ & \ldots \quad] \end{aligned}$$

or

$$V_\pi(\mathbf{x}_0) \leftarrow V_\pi(\mathbf{x}_0) + \alpha_m[d_0 + \gamma d_1 + \gamma^2 d_2 + \ldots] \tag{2.20}$$

where the terms $d_k = r(\mathbf{x}_{k+1}) + \gamma V_\pi(\mathbf{x}_{k+1}) - V_\pi(\mathbf{x}_k)$ are called *temporal differences*. They represent an estimate of the difference at time k between the *expected* cost $V_\pi(\mathbf{x}_k)$ and the *predicted* cost $r(\mathbf{x}_{k+1}) + \gamma V_\pi(\mathbf{x}_{k+1})$. Equation 2.20 is the $TD(1)$ method for calculation of expected costs. A more general formulation is to discount the influence of temporal differences independently of γ by using a factor $\lambda \in [0, 1]$, originating the $TD(\lambda)$ method:

$$V_\pi(\mathbf{x}_0) \leftarrow V_\pi(\mathbf{x}_0) + \alpha_m[d_0 + \gamma\lambda d_1 + \gamma^2\lambda^2 d_2 + \ldots] \tag{2.21}$$

At this stage, one must be careful not to confuse the different roles of γ — which is a discount on future reinforcements, and λ — which is a discount on future temporal differences. The first is usually part of the problem specification, and the second is a choice on the algorithm used to solve the problem.

$TD(\lambda)$ as defined above is not directly implementable, because it is non-causal: future differences $d_1, d_2, \ldots$ are used for updating $V_\pi(\mathbf{x}_0)$. We can fix this up by using the so-called *eligibility traces*, memory variables associated to each state, defined as:

$$e_t(\mathbf{x}) = \begin{cases} \lambda\gamma e_{t-1}(\mathbf{x}) & \text{if } \mathbf{x} \neq \mathbf{x}_t \\ \lambda\gamma e_{t-1}(\mathbf{x}) + 1 & \text{if } \mathbf{x} = \mathbf{x}_t \end{cases}$$

We can then produce a *causal* implementation of $TD(\lambda)$ that is equivalent to its non-causal counterpart (equation 2.21). All the eligibility traces are reset to zero at startup, and then at any time $t > 0$, the learning agent:

1. Visits state $\mathbf{x}_t$ and selects action a_t (according to policy π).
2. Receives from the process the reinforcement $r(\mathbf{x}_t, a_t)$ and observes the next state $\mathbf{x}_{t+1}$.
3. Assigns $e_t(\mathbf{x}_t) = e_t(\mathbf{x}_t) + 1$
4. Updates the costs $V_t(\mathbf{x})$ for all $\mathbf{x}$ according to:

$$V_{t+1}(\mathbf{x}) = V_t(\mathbf{x}) + \alpha_t[r(\mathbf{x}_t, a_t) + \gamma\hat{V}_t(\mathbf{x}_{t+1}) - V_t(\mathbf{x}_t)]e_t(\mathbf{x}) \tag{2.22}$$

5. For all $\mathbf{x}$ do $e_t(\mathbf{x}) = \gamma\lambda e_t(\mathbf{x})$

6. Repeats steps above until a stopping criterion is met;

A variation that has been reported to yield superior results in some problems consists in the use of *replacing traces* $e_t(\mathbf{x}_t) = 1$ (instead of $e_t(\mathbf{x}_t) = e_t(\mathbf{x}_t) + 1$) in step 3 (Singh and Sutton, 1996).

Regardless of causality, $TD(\lambda)$ has been proved to converge to the correct discounted cost (both causal and non-causal realizations) provided some conditions are met (Jaakkola et al., 1994), the most important of those being that all the states must be experienced an infinite number of times. The usefulness of $0 \leq \lambda < 1$ has been verified in practice (*e.g.*, (Tesauro, 1992))and demonstrated analytically in some very simple cases (Singh and Dayan, 1996), but no consistent general results are available. Very often, $TD(0)$,

$$V_\pi(\mathbf{x}_k) \leftarrow V_\pi(\mathbf{x}_k) + \alpha_m[r(\mathbf{x}_k) + \gamma V_\pi(\mathbf{x}_{k+1}) - V_\pi(\mathbf{x}_k)] \tag{2.23}$$

is adopted as a starting point for both theoretical studies and practical implementation.

$TD(\lambda)$ is a robust mechanism for the evaluation of a given policy, and can be used in combination with the Successive Approximation operator for an implementation of an iterative Policy Iteration method. However, there is no model-free TD equivalent for a policy improvement as provided by the Successive Approximation operator:

$$\pi^{k+1}(\mathbf{x}) = \arg\min_a [r(\mathbf{x}, a) + \gamma P(\mathbf{y}|\mathbf{x}, a) V^{\pi^k}(\mathbf{y})] \tag{2.24}$$

This motivates us to consider Value Iteration (instead of Policy Iteration) as a basic DP technique over which it would be possible to develop a *completely* model-free action policy learning algorithm. The next section shows that there is such an algorithm.

2.6.2 Q-Learning

Temporal Differences as discussed above is a method for calculating the expected costs *for a given policy* π. Although it can be used for the learning of action policies through the use of model-based iterative versions of the Policy Iteration algorithm, we would like to have a computational method for a *direct* model-free learning of optimal actions.

Q-learning, an ingenious technique proposed by (Watkins, 1989), is an iterative method for action policy learning in autonomous agents. It is based on the action (or Q) value measurement $Q(\mathbf{x}_t, a_t)$, defined as

$$\begin{aligned} Q(\mathbf{x}_t, a_t) &= \mathrm{E}[r(\mathbf{x}_t, a_t) + \gamma V^*(\mathbf{x}_{t+1})] \\ &= r(\mathbf{x}_t, a_t) + \gamma \sum_{\mathbf{x}_{t+1} \in \mathcal{X}} P(\mathbf{x}_{t+1}|\mathbf{x}_t, a_t) V^*(\mathbf{x}_{t+1})] \end{aligned} \tag{2.25}$$

which represents the expected discounted cost for taking action a_t when visiting state $\mathbf{x}_t$ and following an optimal policy thereafter. From this definition and as a consequence of the Bellman's optimality principle, we have

$$\begin{aligned} V^*(\mathbf{x}_t) &= \mathrm{T}V^*(\mathbf{x}_t) \\ &= \min_a [r(\mathbf{x}_t, a) + \gamma \sum_{\mathbf{x}_{t+1} \in \mathcal{X}} P(\mathbf{x}_{t+1}|\mathbf{x}_t, a) V^*(\mathbf{x}_{t+1})] \\ &= \min_a Q(\mathbf{x}_t, a) \end{aligned} \tag{2.26}$$

and thus

$$Q(\mathbf{x}_t, a_t) = r(\mathbf{x}_t, a_t) + \gamma \sum_{\mathbf{x_{t+1}} \in \mathcal{X}} P(\mathbf{x}_{t+1}|\mathbf{x}_t, a_t) \min_a Q(\mathbf{x}_{t+1}, a) \tag{2.27}$$

or

$$Q(\mathbf{x}_t, a_t) = (\mathrm{T}^q Q)(\mathbf{x}_t, a_t) \tag{2.28}$$

where T^q is the *Action Value operator*

$$(\mathrm{T}^q Q)(\mathbf{x}, a) = r(\mathbf{x}, a) + \gamma \sum_{\mathbf{y} \in \mathcal{X}} P(\mathbf{y}|\mathbf{x}, a) \min_u Q(\mathbf{y}, u) \tag{2.29}$$

Notice that this resembles the Value Iteration operator T previously defined, but there is a fundamental difference: here, the $\min$ operator is *inside* the expectation term. Additionally, notice from equation 2.28 that the action value function Q is the fixed point of the operator T^q, in the same sense that the cost function V_π is the fixed point of the operator T_π, for a given action policy π. And the Q values have a remarkable property: they are policy-independent, yet the optimal action for a given state $\mathbf{x}$ can be obtained through $\pi^*(\mathbf{x}) = \arg\min_a[Q(\mathbf{x}, a)]$.

These characteristics ($\min$ operator inside the expectation term and policy independence) allow an iterative process for calculating an optimal action policy (via action values) which is the essence of the *Q-learning algorithm*, defined as follows. At time t, the agent:

1. Visits state $\mathbf{x}_t$ and selects an action a_t.
2. Receives from the process the reinforcement $r(\mathbf{x}_t, a_t)$ and observes the next state $\mathbf{x}_{t+1}$.
3. Updates the action value $Q_t(\mathbf{x}_t, a_t)$ according to:

$$\begin{aligned} Q_{t+1}(\mathbf{x}_t, a_t) &= Q_t(\mathbf{x}_t, a_t) + \\ &\alpha_t[r(\mathbf{x}_t, a_t) + \gamma \hat{V}_t(\mathbf{x}_{t+1}) - Q_t(\mathbf{x}_t, a_t)] \end{aligned} \tag{2.30}$$

4. Repeats steps above until stopping criterion is met;

where $\hat{V}_t(\mathbf{x}_{t+1}) = \min_a[Q_t(\mathbf{x}_{t+1}, a)]$ is the current estimate of the optimal expected cost $V^*(\mathbf{x}_{t+1})$. The greedy action $\arg\min_a[Q_t(\mathbf{x}_{t+1}, a)]$ is the best the agent thinks it can do when in state $\mathbf{x}_{t+1}$, but for the initial stages of the training process it is a good idea to use randomized actions that encourage exploration of the state space. The first three items on the sequence above define the *experience tuple* $\langle \mathbf{x}_t, a_t, \mathbf{x}_{t+1}, r(\mathbf{x}_t, a_t) \rangle$ undergone by the agent at a given instant of time $t + 1$.

Observe that Q-learning assumes a tabular representation in which action values for each state are stored in separate tables, one for each action, as in Figure 2.3.

2.6.3 Why is Q-Learning so Popular?

Q-learning is certainly the most popular RL method. There are some reasons for this. First, it was the pioneering (and up to the moment, the only) thoroughly studied RL method for control purposes, with a strong proof of convergence established

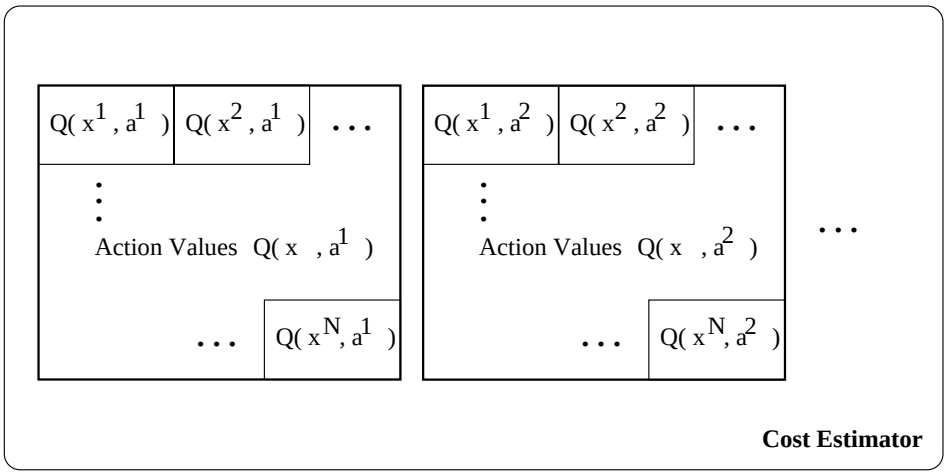

Figure 2.3: Look-up table implementation in Q-learning. Each table separately stores all the action values for each action.

in (Jaakkola et al., 1994)[6] Second, it is the extension *par excellence* of the autonomous learning concept to Optimal Control, in the sense that it is the simplest technique that directly calculates the optimal action policy without an intermediate cost evaluation step and without the use of a model[7]. The third reason for Q-learning's popularity was some early evidence that it performed better than other RL methods (Lin, 1992), (see below for a description of some of these methods). With the recent proliferation of alternative RL algorithms, this evidence is not as clear today as it was in the late Eighties. In fact, many other techniques have a record of successful performance, to be considered with some reserve due to a widespread lack of robust statistical analysis of results.

2.6.4 Alternatives to Q-Learning

An interesting variation for Q-learning is the SARSA algorithm (Sutton, 1996), which aims at using Q-learning as part of a Policy Iteration mechanism. Its iteration loop is based on the action value update

$$\begin{aligned} Q_{t+1}(\mathbf{x}_t, a_t) &= Q_t(\mathbf{x}_t, a_t) + \\ &\alpha_t[r(\mathbf{x}_t, a_t) + \gamma Q_t(\mathbf{x}_{t+1}, a_{t+1}) - Q_t(\mathbf{x}_t, a_t)] \end{aligned} \tag{2.31}$$

Naturally, if a_{t+1} is chosen to be $\arg\min_a [Q_t(\mathbf{x}_{t+1}, a)]$ this algorithm is equivalent to standard Q-learning. SARSA however admits a_{t+1} to be chosen randomly with a predefined probability. Like Q-learning, this method has been proved to converge to the optimal action values, provided that the actions assimptotically approach a greedy policy (Szepesvári, 1997). Because it eliminates the use of the $\min$ operator over the actions, this method is faster than Q-learning for situations where the action set has high cardinality. Moreover, it allows us to consider a SARSA(λ) algorithm, similar to

[6] Actually, a very ingenious proof of convergence was published shortly after Watkins' thesis publication (Watkins and Dayan, 1992), making it the first RL control method to have a solid ground.

[7] Of foremost importance in the popularization of the link between Q-learning and DP was Satinder Singh's PhD thesis (Singh, 1994)

causal $TD(\lambda)$. All the eligibility traces are reset to zero at startup, and then at any time $t > 0$, the learning agent:

1. Visits state $\mathbf{x}_t$ and selects action a_t (according to policy π).
2. Receives from the process the reinforcement $r(\mathbf{x}_t, a_t)$ and observes the next state $\mathbf{x}_{t+1}$.
3. Assigns $e_t(\mathbf{x}_t, a_t) = e_t(\mathbf{x}_t, a_t) + 1$
4. Updates the action values $Q_t(\mathbf{x}, a)$ for all $\mathbf{x}, a$ according to:

$$Q_{t+1}(\mathbf{x}, a) = Q_t(\mathbf{x}, a) + \alpha_t[r(\mathbf{x}_t, a_t) + \gamma Q_t(\mathbf{x}_{t+1}, a_{t+1}) - Q_t(\mathbf{x}_t, a_t)]e_t(\mathbf{x}, a) \tag{2.32}$$

5. For all $\mathbf{x}, a$ do $e_t(\mathbf{x}, a) = \gamma\lambda e_t(\mathbf{x}, a)$
6. Repeats steps above until a stopping criterion is met;

Other attempts at combining Q-learning and $TD(\lambda)$ are the Q(λ) (Watkins, 1989) and Peng and William's algorithms (Peng and Williams, 1996).

Many successful applications of RL are simulation-based (or model-based), in the sense that state sequences can be simulated before an actual experience takes place. There is therefore a model from which those simulations originate, although it does not imply an explicit use of transition probabilities (the model can be slowly built up as experimentation progresses). Such methods have a potential capability of greatly reducing the huge computational requirements of DP, whilst at the same time making better use of experience than standard TD methods.

The Dyna-Q algorithm, proposed by (Sutton, 1990), is characterized by the iterative learning of a direct model for the transition probabilities and rewards, simultaneously with the application of a DP method to compute the action values and action policy. Given an experience tuple $\langle \mathbf{x}_t, a_t, \mathbf{x}_{t+1}, r(\mathbf{x}_t, a_t)\rangle$, Dyna-Q updates a model $\tilde{P}(\mathbf{x}_{t+1}|\mathbf{x}_t, a_t)$ for the transition probabilities and only then updates a policy for the agent using one step of value iteration to approximate the solution of equation 2.27:

$$Q(\mathbf{x}_t, a_t) \leftarrow r(\mathbf{x}_t, a_t) + \gamma \sum_{\mathbf{x}_{t+1}} \tilde{P}(\mathbf{x}_{t+1}|\mathbf{x}_t, a_t) min_a Q(\mathbf{x}_{t+1}, a) \tag{2.33}$$

Finally, k additional similar updates are made on randomly chosen states. A new action is then chosen as in standard Q-learning.

Notice that Dyna-Q is an *on-line* learning method, because the agent does not wait until the transition probabilities are correctly estimated. It uses information more efficiently than standard Q-learning because it makes more than a single update per stage, at the expense of additional storage for the estimated transition probabilities. There are some variants for Dyna-Q that make a more efficient choice of the k states chosen for update (Moore and Atkeson, 1993).

2.6.5 Other RL techniques: A Brief History

RL techniques have been proposed since at least the late fifties, long before the Temporal Differences theory had been developed. The diversity of approaches testifies to the importance and complexity of the basic autonomous learning concept. This section

briefly presents some methods that are not direct applications of the TD method. Some of them, however, are true precursors of Temporal Differences for being also motivated by the idea of successive predictions.

The BOXES algorithm (Michie and Chambers, 1968) was one of the earliest attempts to solve control problems in a model-free context. Its basic motivation is the fact that it may be advantageous to decompose a large task into a collection of small subtasks, each of them represented by a partition (box) in the state space of the original problem.

The idea of using a collection of boxes as a discretized version of the original problem is reminiscent of Q-learning, but BOXES is *not* a TD method. It bases learning on *life* terms $L(\mathbf{x}, a)$, functions of the elapsed time between use and failure of a given action-box pair $(\mathbf{x}, a)$ [8]:

$$L(\mathbf{x}, a) = \sum_{i=0}^{n} (T_{final} - T_i(\mathbf{x}, a)) \tag{2.34}$$

where T_{final} is the complete duration of the trial and the $T_i(\mathbf{x}, a)$ are the times at which the pair $(\mathbf{x}, a)$ have been visited. Only *after* the trial is over, *life-time* terms LT — which ultimately are responsible for the action choice — are changed according to the update rule:

$$LT(\mathbf{x}, a) \leftarrow \gamma LT(\mathbf{x}, a) + L(\mathbf{x}, a) \tag{2.35}$$

where $0 \leq \gamma < 1$ is a discount factor. Notice that LT is a discounted accumulation of past L terms.

BOXES was proposed almost thirty years ago, but already at that time its developers had interesting insights about autonomous learning. As a matter of fact, the algorithm has a mechanism for action selection that encourages exploration, correctly stressed by the authors as an important issue. The technique, however, always suffered from a lack of formal development, ironically due to its own novelty: The update rules are empirical, and more recent developments (Sammut, 1994) (McGarity et al., 1995) have not tackled the possibly complex role of the algorithm parameters.

A performance comparison between BOXES and AHC in the cartpole problem was highly favorable to the Temporal Differences method (Barto et al., 1983).

Another interesting development not often mentioned in literature is the Adaptive Optimal Controller by Witten (Witten, 1977). Studying the problem of designing a learnable controller able to minimize a discounted reward of the form $(1 - \gamma) \sum_{t=0}^{r} \gamma^t r(\mathbf{x}_t)$, Witten managed to prove some convergence theorems for learning rules similar to the ones used in TD methods. The theorems, however, are all about convergence of the mean for the discounted costs, *i.e.*, they establish the convergence to zero of the error between the *mean* value of cost estimates and the real expected costs. Moreover, Witten did not fully analyze the problems caused by policy changes, and considered only small perturbations of the cost updating process. The conclusions were pessimistic: if the cost estimates did not converge rapidly, the action policy could deteriorate. As showed much later, this is not the case for Q-learning. Nevertheless, Witten's development was insightful and was possibly the first attempt to formally link RL with the theory of Markov Decision Processes. Its limited success only accentuates the real breakthrough that was provided by the development of TD methods.

[8] It might be possible in principle to use measures of performance other than elapsed control time.

The Bucket Brigade algorithm (Holland and Reitman, 1978), originally proposed for solving the credit assignment problem in classifier systems, is another example of a technique to solve the credit assignment problem. Classifier systems are parallel rule-based systems that learn to perform tasks through rule discovery based on genetic algorithms (Dorigo and Colombetti, 1994). Rules are represented by classifiers, each of which is composed by a *condition* part and an *action* part. Input messages ('states' of the external process) are compared to all conditions of all classifiers, and the messages specified by the action part of all the matching conditions are then put in a message list that represent the 'agent's' current output. The many rules are generated randomly, but are then filtered out and mutated through genetic operations that select 'the most fit', *i.e.*, the ones that produce the best output messages. Assessing the fitness of the classifiers is the RL task involved.

The Bucket Brigade algorithm operates as follows: When a matching classifier C places its message on the message list, it pays for the privilege by having its *strength* $S_t(C)$ reduced by a *bid* $B_t(C)$ proportional to $S_t(C)$:

$$S_{t+1}(C) = S_t(C) - B_t(C) \tag{2.36}$$

Then, the classifiers C' that sent the message matched by C have their strength increased by the shared amount $a\,B_t(C)$ of the bid:

$$S_{t+1}(C') = S_t(C') + aB_t(C) \tag{2.37}$$

The strength represents the fitness or expected reward associated with a given classifier. The algorithm operates by propagating the bids backwards in time, and as these bids are themselves proportional to the strengths, there is a temporal credit assignment process in the sense that sequences of messages associated with high final payoffs (reinforcements), eventually added to the rules that led to them, will tend to be defined by the corresponding classifiers.

A comparison between a version of the Bucket Brigade algorithm and Q-learning in a single example task gave inconclusive results (Matarić, 1991).

There are other developments in RL theory and practice that deserve mention. Probably the first RL technique to appear in the literature was the checkers playing machine developed by (Samuel, 1959), which punishes or rewards moves according to the difference between successive position evaluations, heuristically computed. A kind of non-discounted RL method is the *Selective Bootstrap Adaptation* network (Widrow et al., 1973), that uses modified ADALINE elements (Widrow and Hoff, 1960) whose weights are changed according to a RL process divided in two phases. In the first, the control task is performed without any weight modification. In the second, the control task is retried with the weights updated to reward or punish the corresponding outputs, depending on the overall performance computed in the first phase. The method is then similar to BOXES in the sense that updates are made only after the end of the control trial. Finally, (Saerens and Soquet, 1989) suggested a model-free neural controller based on qualitative information about the rate of variation of the process output with respect to the agent's actions. The authors reported good results in the cartpole task, but stressed that handcrafted rules were necessary to keep the cart within the track boundaries.

Chapter 3

Experience Generalization

We now introduce RL agents to one of the problems of the real world: the large number of states and action possibilities encountered in many applications. This dimensionality problem forces the agent to be more audacious in its experience updating process, by producing modifications not just for a single state-related experience per iteration, but also to other states, through a process called generalization. Moreover, it is well possible that the state space can be so large that representing all the possible states (let alone producing explicit modifications on its related costs) is infeasible. This brings us to the problem of function approximation, which implies compact representations of states for which generalization is an *emergent* property. Finally, there is the problem of how to pinpoint state features upon which a similarity metric for the generalization process is defined. This motivates the use of online, experience dependent clustering or splitting of states, discussed at the end of this chapter.

3.1 The Exploration/Exploitation Tradeoff

One of the necessary conditions upon which RL algorithms can find an optimal action policy is the complete exploration of the state space, normally infeasible in practical situations. When control and learning are both at stake, the learning agent must try to find a balance between the *exploration* of alternatives to a given policy and the *exploitation* of this policy as a mechanism for assessing its associated costs. In other words, it must consider that trying unknown alternatives can be risky, but that keeping the same policy *ad infinitum* will never lead to improvement.

This tradeoff between caution and probing, between control objective and parameter estimation objective is a well-known issue in Optimal Control theory and is commonly referred to as the *dual control problem* (Bertsekas, 1995b). RL methods usually include a stochastic action selector that allows some additional exploration of the state space, but this is not enough for reasonably large state spaces (*i.e.*, spaces with a large number of discretized states), unless a very long training time is assumed.

3.2 Accelerating Exploration

The exploration/exploitation tradeoff would not be a problem of practical importance if exploration of the state space could quickly provide reasonable estimates for costs or

action values. Unfortunately, this is not the general case, as usually multiple visitations to every single state are necessary before a good action policy can be obtained. We can then say that the spatial and temporal locality of RL methods is inefficient: it makes bad use of the experience by slightly updating only the cost or action value associated with the corresponding visited state.

Two approaches are possible for improving the performance of RL methods with respect to the exploration problem: a) generalizing the experience obtained from single updates using predefined features or b) iteratively clustering or splitting states and thus concentrating resources on 'interesting' parts of the state space.

3.3 Experience Generalization

Experience generalization can occur in two contexts. The first is when a look-up table representation of the state space is possible, but even so a desired level of performance is difficult to attain. These are the cases when the agent is subject to a time constraint for the learning process which does not give it a chance to frequently experiment every single state transition, or when there is a *experience cost* that limits experimentation (this is often the case for real robots). In these situations, an embedded mechanism that generalizes the consequences of real experiences to those that have not been sufficiently experimented can be useful.

The second context is when compact function approximation schemes are necessary due to an extremely large number of states that makes a look-up table implementation of the state space impossible. Typically, these approximations are implemented by function approximators that associate states or state features with costs or action values. In this case, experience generalization is an *emergent* property of the approximation scheme.

The generalization issue in RL can be seen as a *structural* credit assignment problem. It implies *spatial* propagation of costs across similar states, whilst temporal credit assignment — as performed by standard RL algorithms — implies *temporal* propagation of costs. Figure 3.1 illustrates these concepts. Generalization is a complex problem because there is no 'Spatial Difference' learning algorithm: the feature that defines how structurally similar states are with respect to cost (or action value) expectations is precisely the optimal cost (action value) function, obviously not available in advance. For standard Temporal Differences acting exclusively on the time dimension, similarity between sequentially visited states is defined by the discount factor — which is given in advance as a parameter of the problem — and by the frequency of experiences, which is assessed online.

3.3.1 Experience Generalization in Lookup Tables

In this case, states can be represented in a lookup table and there is model-free learning of action policies. Cost (or action value) generalization can be controlled along time, a characteristic that is convenient because measures of state similarity frequently cannot be chosen adequately and a convenient sampling of states cannot be defined in advance. Notice that the possibility of representing the state space in tabular form makes experience generalization a *purposeful* property, and not an emergent characteristic.

In one of the first studies on RL applied to real robotic navigation (Mahadevan and Connell, 1992), the authors observed the structural credit assignment problem and proposed a generalisation solution to it based on the fact that 'similar' sensed states must

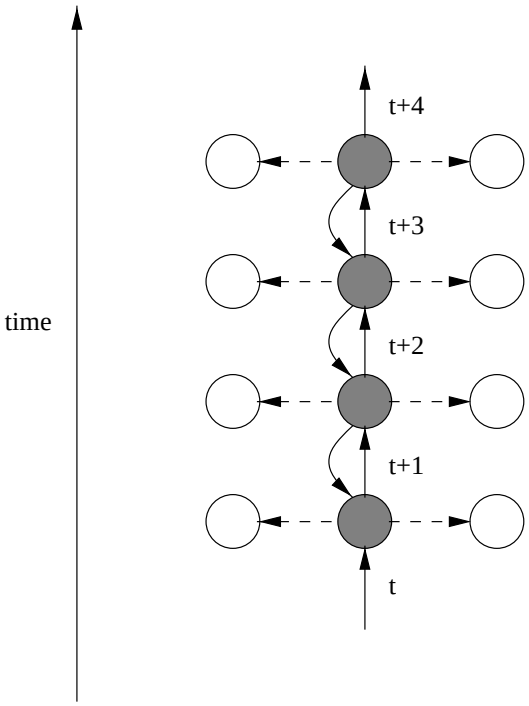

Figure 3.1: Structural and temporal credit assignment. The dark circles are states visited in a temporal sequence $t, t+1, t+2, t+3, t+4$. The curved arrows represent propagation of temporal updates, and the dashed arrows represent propagation of the structural credit assignment among similar states that are not visited in sequence.

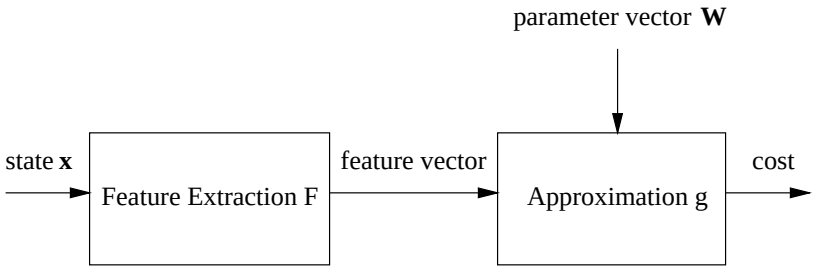

Figure 3.2: Feature-based compact representation model.

have 'similar' costs. They defined a weighted Hamming distance for the calculation of this similarity based on a previously assessed relative importance of sensors. The reported experiments showed that the use of this hand coded approach led to results similar to the ones obtained when statistically based clustering (described near the end of this chapter) was used. However, as the main concern was experimental evaluation of RL methods in modular tasks, no formal or deeper studies on the generalization issue were made.

An extension of this concept is the use of *softly* aggregated states in Q-learning, as proposed in (Singh et al., 1995). In this case, sets defined by groups of similar states are not disjoint, but defined by probability distributions. Error bounds for the resulting approximations can be obtained for this algorithm, and a recent result (Szepesvári and Littman, 1996) shows that any form of compact representation based on tables in a space partitioned by sets of similar states actually converges, provided that the sampling of state-action pairs comes from a fixed distribution or, more generally, if they are ergodically sampled[1]. However, the asymptotic values depend both on the similarity metric *and* on this sampling, and may not be close to the optimal ones. A way out of this problem is to consider a temporally 'vanishing' similarity (or a slow state deaggregation) so that standard RL (over the real states) acts on later stages of training (Ribeiro, 1998; Ribeiro and Szepesvári, 1996).

3.3.2 Experience Generalization from Function Approximation

The problem of how to generate useful generalization when compact representations are necessary is an extremely complex one for RL algorithms. A suitable model for the general function approximation problem is the *feature-based compact representation* (Tsitsiklis and Roy, 1996), illustrated in figure 3.2. It divides function approximation into two stages. The first one is the feature extraction phase, where each state $\mathbf{x} \in \mathcal{X}$ is mapped to a hand crafted feature vector $F(\mathbf{x}) = (f_1(\mathbf{x}), \ldots, f_k(\mathbf{x}))$ that encodes the relevant properties of $\mathbf{x}$. A feature is basically a discrete measure of similarity, and this first stage does exactly what a generalization scheme for look-up table representations does. If the new space defined by the feature vectors is still too large to allow a tabular representation, a second phase that employs a compact parametric function approximator is necessary. This approximator can be a linear or nonlinear architecture that generates an approximation $\tilde{V}(\mathbf{x}, \mathbf{W}) = g(F(\mathbf{x}), \mathbf{W})$ for the costs or action values.

[1] *I.e.*, if state-action pairs $(\mathbf{x}, a)$ are sampled assimptotically according to fixed distributions $p^{\infty}(\mathbf{x}, a)$ ($P(\mathbf{x}_t = \mathbf{x}, a_t = a)$ converges to $p^{\infty}(\mathbf{x}, a)$).

Figure 3.3: A two-states Markov chain for which a least-squares approximation method for cost calculation may diverge.

Function approximators such as artificial neural networks have been used in RL for a long time, specially for cost evaluation. Initial developments such as an expert-level Backgammon playing TD-based neural network (Tesauro, 1992; Tesauro, 1995) were very encouraging (see discussion at the end of this section), but disastrous results in simple domains were also reported (Boyan and Moore, 1995). The trouble is that generalisation in DP (and, by extension, in RL) is not a simple approximation problem where given a set of 'training' instances $\{(\mathbf{x}_1, V^*(\mathbf{x}_1)), (\mathbf{x}_2, V^*(\mathbf{x}_2)), \ldots, (\mathbf{x}_k, V^*(\mathbf{x}_k))\}$ a convenient interpolator must be found, simply because the optimal costs V^* are not available in advance. Consider what would be a straightforward attempt of using a standard 'least-squares' fit method for the parameter vector $\mathbf{W}$ of a cost approximator $\hat{V}(\mathbf{x}, \mathbf{W})$:

$$\mathbf{W}_{t+1} = \arg\min_{\mathbf{W}} \sum_{\mathbf{x}} (\hat{V}(\mathbf{x}, \mathbf{W}) - \mathrm{T}\hat{V}(\mathbf{x}, \mathbf{W}_t)^2 \tag{3.1}$$

as an approximation scheme to what could be a least-squares method if the optimal costs V^* were available:

$$\mathbf{W}_{t+1} = \arg\min_{\mathbf{W}} \sum_{\mathbf{x}} (\hat{V}(\mathbf{x}, \mathbf{W}) - V^*(\mathbf{x}))^2 \tag{3.2}$$

This seems to make sense because each iteration in this case approximates the standard Value Iteration method described in chapter 2. Nonetheless, the following counterexample — originally described in (Bertsekas, 1995a) — shows that even if the states are perfectly discriminated by this approximation scheme, divergence can take place.

Consider the simple two-states Markov chain illustrated in figure 3.3. There is no control policy and every state reward is 0, so the optimal cost V^* associated to each state x^1 and x^2 is also 0. Consider a linear function approximator of the form

$$\hat{V}(x, w) = wf(x), x \in \{x^1, x^2\} \tag{3.3}$$

where the feature f is such that $f(x^1) = 1$ and $f(x^2) = 2$.

Notice that if $w = 0$ the correct optimal cost is obtained, so a perfect representation is possible. Applying one step of the 'least-squares' fit method 3.1 leads to

$$\begin{aligned} w_{t+1} &= \arg\min_{w} \sum_{x} (\hat{V}(x, w) - \mathrm{T}\hat{V}(x, w_t)^2 \\ &= \arg\min_{w} ((w - \gamma 2w_t)^2 + (2w - \gamma 2w_t)^2) \end{aligned} \tag{3.4}$$

Hence,

$$w_{t+1} = \frac{6}{5}\gamma w_t \tag{3.5}$$

and divergence takes place if $\gamma > 5/6$.

Complications such as this seems to motivate the use of look-up table representations whenever possible, specially because no strong theoretical results for nonlinear compact representations are available. If the number of state features is small enough, a particular class of approximators can be defined by assigning a single value to each point in feature space, in such a way that the parameter vector $\mathbf{W}$ has one component for each possible feature vector. This corresponds to a split of the original state space into disjoint sets $S^1, S^2, \ldots, S^m$ of *aggregate states*, each set ideally representing states that should have the same optimal costs. In this case, g acts as a hashing function, and we thus have an alternative view of the algorithms considered in the previous section. Formally, $\tilde{V}(\mathbf{x}, \mathbf{W}) = w^j, \forall \mathbf{x} \in S^j$, and a natural update rule for the cost parameters is

$$w_{t+1}^j = w_t^j + \alpha_t^j (\mathrm{T}\hat{V}(\mathbf{x}, \mathbf{W}) - w_t^j) \tag{3.6}$$

where $\alpha_t^j = 0$ if $x_t \notin S^j$.

Independently of how the states are sampled, it is possible to define theoretical worst case bounds for the resulting approximations which depend on the quality of the chosen features (Tsitsiklis and Roy, 1996). Similar bounds can be obtained for some classes of linear function approximators and for the Q-learning algorithm (Bertsekas and Tsitsiklis, 1996). As we mentioned before, however, an additional dependency on the state-action sampling is present in this last case.

Yet, we must stress that, even considering all the possible drawbacks of function approximation applied to RL, a very careful feature selection and the availability of a simulation model can lead to impressively successful results. Among those successes stands TD-Gammon, the aforementioned TD-based feedforward neural network that autonomously learn how to play Backgammon at expert level (Tesauro, 1995). TD-Gammon produces a mapping from board state to corresponding expected reward, for a given action policy. It thus corresponds to an on-line version of the Policy Evaluation mapping. Learning of action policies is then carried out in the following way: for each board state $\mathbf{x}_t$, all the possible actions are tried off-line and the one that gives the largest expected reward for the new state $\mathbf{x}_{t+1}$ is chosen. The difference between the expected reward for $\mathbf{x}_{t+1}$ and the one for $\mathbf{x}_t$ is then used to change (through gradient descent) the weights of the network for the input $\mathbf{x}_t$. Pseudocode that can be used for implementing a version of this algorithm (and for any other implementation combining TD and the standard error backpropagation method in a neural network) can be found at:

- ftp://ftp.cs.umass.edu/pub/anw/pub/sutton/td-backprop-pseudo-code

Actually, there are many TD variants that use a simulation model to compare and approximate future states with respect to cost evaluation, and the best practical results on RL have actually been obtained with this approach. For instance, a TD approximating architecture for channel allocation in cellular telephony systems that outperforms many commonly used heuristic methods has been recently reported (Singh and Bertsekas, 1997). A Java demo for it can be found at:

- http://www.cs.colorado.edu/ baveja/Demo.html

Another architecture was reported to solve a complex elevator scheduling problem (Crites, 1996). Even though the idea of using off-line experience to simulate model

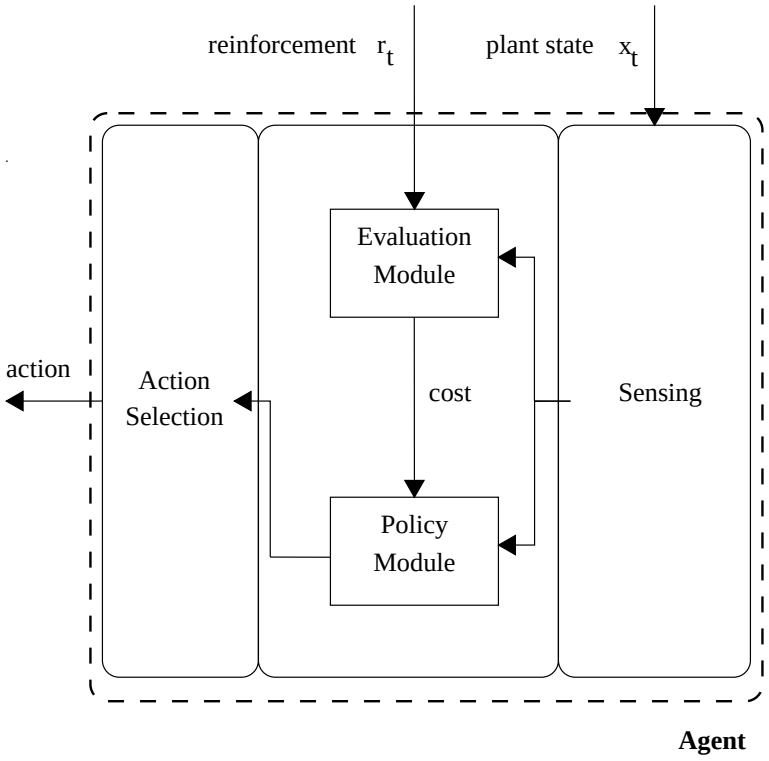

Figure 3.4: An Adaptive Heuristic Critic (AHC) agent. The Policy Module generates improved actions for a given policy using the costs estimated by the Evaluation Module.

behavior violates the basic autonomous learning principles, its applicability to engineering situations where this model is available is indisputable and is demonstrated by these and other examples. A comprehensive review on the subject can be found in (Bertsekas and Tsitsiklis, 1996).

The Adaptive Heuristic Critic

Let us now consider with special attention the first TD-based technique proposed for model-free action learning and function approximation: the Adaptive Heuristic Critic (AHC) method. Its first results on the task of learning to control the cartpole system (Barto et al., 1983) draw large interest to the TD approach. Unlike Q-learning, AHC is an actual attempt at a model-free Policy Iteration mechanism: it is better understood as an attempt to combine the policy evaluation step with an on-line action selection strategy. Despite having largely unknown properties with respect to convergence conditions, AHC works well in many situations.

An AHC architecture is illustrated in Figure 3.4. It has two main components: an *Evaluation Module* which computes an approximation of the cost function $V(.)$ for every input state, and a *Policy Module* which iteratively tries to estimate an optimal action policy $\pi^*(.)$. An stochastic action selector is used to generate exploratory random actions at the beginning of the training process. During learning, both the evaluation and policy modules are adjusted. The evaluation module operates as in the simulation-based methods we mentioned in the last section: for any given state, its

parameters are updated by comparing the current output $V(\mathbf{x}_t)$ with the expected cost $r(\mathbf{x}_t, a_t) + \gamma V(\mathbf{x}_{t+1})$. The action that led to $\mathbf{x}_{t+1}$, however, is not determined through off-line comparison between the cost functions obtained from the application of different actions. Instead, the action is also computed on-line as the output of the policy module. The parameters of the policy module, on its turn, are updated according to a gradient method which encourages or discourages the current action, depending if the updated $V'(\mathbf{x}_t)$ is smaller or larger than the former $V(\mathbf{x}_t)$. The underlying idea is that if the updated cost is smaller, than the corresponding action should be encouraged because it decreases the expected cost for the visited state. Notice that the policy network must not be updated with respect to other actions, since from a single experience nothing is known about their merits.

Separately setting up the right learning parameters in the AHC method can be very difficult, as the evaluation and policy modules perform tasks that are actually conflicting. The evaluation module needs as much *exploitation* of a given action policy as possible, in order to correctly assess it. On the other hand, the policy module needs as much *exploration* of the state space as possible, in order to find out the actions that lead to the 'best' parts of the state space. However, the assessment of these actions depend on good cost evaluations, that must be actually provided by the evaluation module.

The CMAC

An attempt at solving the generalization problem that involves a tabular representation, but without the need of a memory position for every state-action pair was proposed in the same report where Q-learning was firstly put forward (Watkins, 1989), and consists of the use of a sparse coarse coded approximator, or CMAC (Albus, 1971) (Thomas Miller, III et al., 1990). A CMAC is represented by a) a set of overlapping tilings, each of them corresponding to a collection of disjoint tiles that partition the state space, b) an array $\mathbf{U} = \mathbf{u}[1], \ldots, \mathbf{u}[n]$ of scalar or vector values adjusted incrementally, and c) a hash function H which maps each tile to an element of $\mathbf{U}$. The tilings are arranged in such a way that each point in the state space corresponds to a collection of tiles, each from a different tiling.

In the context of Q-learning, the elements of the array $\mathbf{U}$ actually share the action values for the state-action pair $(\mathbf{x}, a)$, encoded by the associated tiles. If, say, 5 tiles $t_1, \ldots, t_5$ are selected, H associates them with the corresponding array elements $k_1 = H(t_1), \ldots, k_5 = H(t_5)$, and the action value is calculated as:

$$Q(\mathbf{x}, a) = (\mathbf{u}[k_1] + \ldots + \mathbf{u}[k_5])/5 \tag{3.7}$$

During learning, all the array elements associated with the visited state are updated accordingly:

$$\mathbf{u}[k_1] \leftarrow \alpha[r(\mathbf{x}, a) + \gamma \hat{V}(\mathbf{y}) - Q(\mathbf{x}, a)] \tag{3.8}$$

Figure 3.5 illustrates the mappings involved in the CMAC approach. The advantages of this method are its computational speed and simplicity, coupled with a local generalization capability (a limited set of tiles is 'active' for a given input, provided the number of elements in the array $\mathbf{U}$ is large enough). It has been argued (Sutton, 1996) that this locality leads to more successful results than global approximators (*e.g.*, neural networks), but it is clear that even local generalization can cause problems if it is not properly controlled.

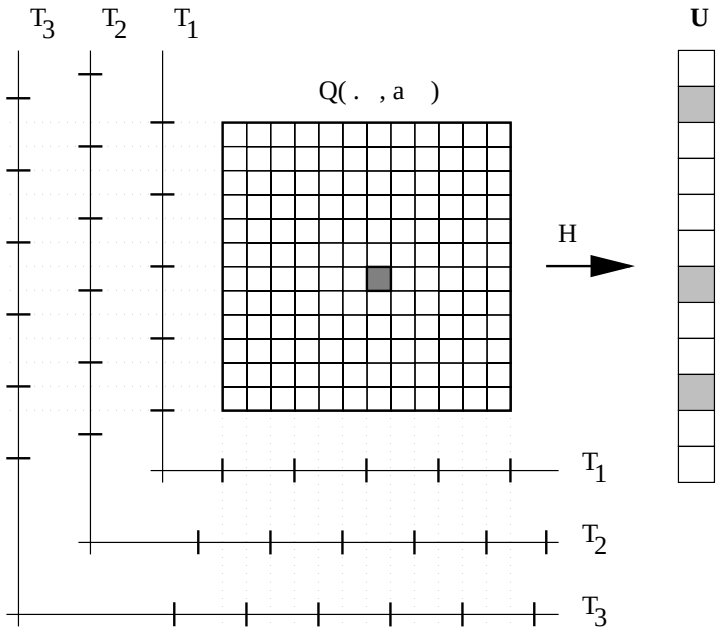

Figure 3.5: The CMAC approach. Each tiling (in this case, T_1, T_2 and T_3) has a single tile associated with the state of the process. The hash function H then maps the tiles to different positions in U.

3.4 Iterative Clustering (Splitting)

When convenient features for state aggregation are not available at all, there is the possibility of trying to find similarities among states through a clustering or expanding method based on a statistical analysis carried out in parallel with the training process. Actually, clustering was originally suggested as an alternative to handcrafted encoding in (Mahadevan and Connell, 1992), which proposed a method founded on empirically tested rules based on similarity among state features or action values (as computed by the learning algorithm). A tradeoff between these two criteria was also obtained empirically. A dual development is the G algorithm (Chapman and Kaelbling, 1991), that incrementally builds up a tree-structured action value table. It begins by supposing that all the input features are relevant (generalisation over the whole state space), collapsing the entire look-up table into a single action value. By collecting statistics about the importance of individual features, the table is gradually split into different subspaces corresponding to the presence or absence of each analyzed feature.

Methods based on statistical analysis can have problems. Apart from possibly mistaken assumptions for the data sampling distribution in the statistical tests, an information gathering process can be in trouble if the action values change quicker than irrelevant input features. These can then appear as important because of the comparatively large variation of the corresponding action values for small feature changes, as recorded by the statistical collection. The problem was originally noticed in (Chapman and Kaelbling, 1991), which then suggested dividing the learning process into action value and feature relevance phases. Estimated Q values are held constant whilst relevance statistics are collected, with a phase switching mechanism based on additional statistical analysis.

While some success has been obtained by statistical approaches, only studies in very specific domains have been described. In any case, no relative advantages over hand crafted encoding have been reported (Mahadevan and Connell, 1992), and the statistical gathering process usually makes the learning process extremely slow (Chapman and Kaelbling, 1991).

Chapter 4

Partially Observable Processes

Apart from the need for generalization, there is a second issue that is extremely important if RL is to be applied in practical problems: partial state observability. Any researcher or engineer is aware that perfect state observation is a concession usually found only in basic Control Theory textbooks. Realistic applications involve systems that are observed through noisy, unreliable and uninformative sensors, which give only rough indications about the true state of the dynamic process to be controlled. Partial observability implies non-Markov observations, destroying one of the theoretical basis for the 'good behavior' of RL algorithms.

4.1 Partial Observability and DP

It is illustrative to consider briefly how the concept of partial observability is dealt with in the field of Optimal Control and how DP techniques can be directly adapted to the case.

An observation $\mathbf{o}_t$ made by the agent at time t is typically some function of the state of the process $\mathbf{x}_t$, last action a_{t-1} and random disturbance v_t:

$$\mathbf{o}_t = h(\mathbf{x}_t, a_{t-1}, v_t) \tag{4.1}$$

As the observations are less informative than the states, the only straightforward way of summarizing all the available information about the process is to consider the whole history of past observations. This is done by considering an *information vector* $\mathbf{I}_t$ defined as

$$\mathbf{I}_t = (\mathbf{o}_0, \mathbf{o}_1, \ldots, \mathbf{o}_t, a_0, a_1, \ldots, a_{t-1}) \tag{4.2}$$

and by restating the control algorithms accordingly. In fact, the information vectors define a dynamic process

$$\mathbf{I}_{t+1} = (\mathbf{I}_t, a_t, \mathbf{o}_{t+1}) \tag{4.3}$$

where the 'state' is the information vector itself and the new observation $\mathbf{o}_{t+1}$ is the 'disturbance'. Furthermore, this new process is Markov, since

$$P(\mathbf{I}_{t+1}|\mathbf{I}_t, \mathbf{I}_{t-1}, \ldots, a_t) = P(\mathbf{I}_{t+1}|\mathbf{I}_t, a_t) \tag{4.4}$$

Combined with some standard restrictions on the disturbances, this formulation allows for the development of DP techniques very similar to the ones presented in chapter 2. However, this has more theoretical than practical relevance: even for very simple problems, the dimension of the information vector space can be overwhelming, and only in a very particular case (linear state equations and quadratic costs) is an analytical solution possible (Bertsekas, 1995b). The practical inconvenience of this general formulation has motivated a series of different suboptimal approaches to the problem of controlling partially observable processes.

4.2 Partial Observability and RL

The huge literature and the extensive research on control of partially observable processes testify to the complexity of the problem. It is not surprising that things get even more complicated once the restrictive conditions of Reinforcement Learning are added.

The fundamental complication is a consequence of the credit assignment practice itself. Even if a given observation $\mathbf{o}_t$ perfectly defines a state, a subsequent observation $\mathbf{o}_{t+n}$ may be ambiguous and then propagate (through $r(\mathbf{x}_t) + \gamma \hat{V}(\mathbf{o}_{t+1})$) wrong updates. The estimated cost for $\mathbf{o}_t$ may then become wrong, and then propagate again to all the previous states. This 'contamination' process can affect many observations, producing the *perceptual aliasing problem* (Whitehead and Ballard, 1990).

4.2.1 Attention-Based Methods

One of the greatest conceptual breakthroughs made by the recent developments on self-improving agents and autonomous learning is the emphasis on the physical interaction agent-process and on the learning agent (and not the engineer) as the designer of action policies. This difference of perspective is clear as terms such as 'partially observable process' — which emphasizes a process condition instead of an agent constraint — get routinely used. Additionally, this change of orientation towards the controller accentuates possible properties not yet fully considered in traditional Control theories. One of these properties is the agent's capability to control its observations through an active (or attentional) vision mechanism.

The problem of active vision has been usually formulated as a search for perceptual policies for recognition of given objects (Kappen et al., 1995; Blake and Yuille, 1992). This is a complex problem in itself, and as pointed out in a classical paper by Brooks, studies on this issue in the context of autonomous learning were almost completely ignored until a few years ago (Brooks, 1991).

Adapting the classical formulation to the case of active vision agents, an attentional mechanism defines an observation as a function of the state of the process $\mathbf{x}_t$, last control action a_{t-1} and random disturbance v_t, plus an *attentional setting* $b_t \in \mathcal{B}$:

$$\mathbf{o}_t = g(\mathbf{x}_t, a_{t-1}, b_t, v_t) \tag{4.5}$$

The attentional setting corresponds to a perceptual action which defines how the agent chooses its observations. The concept of 'active' vision implies a free choice (by the agent) of perceptual actions. Thus,

$$b_t = f(\text{ internal condition of the agent at time } t) \tag{4.6}$$

Notice that b_t has a different nature than a_t as it does not disturb the actual state of the process.

Attempts to combine learning of perceptual and control actions have often been characterized by some very specific restrictions (Whitehead and Lin, 1995)[1]:

- At each time step, control is divided into two phases: a perceptual stage and an action stage. The perceptual stage aims to generate internal representations that are Markov, whilst the action stage aims to generate optimal control actions.
- Learning for the action stage can be done with common RL methods. Learning for the perceptual stage is based on the monitoring of ambiguous (non-Markov) internal states.
- It is assumed that the external state can always be identified from immediate sensory inputs. In this sense, such problems are not about strictly partially observable processes, because there is always at least one attentional setting that can discriminate the correct state.

Two representative active vision methods are the G algorithm (Chapman and Kaelbling, 1991), described in the last chapter and that can be also seen as a method for voluntary feature selection, and the Lion algorithm (Whitehead and Ballard, 1990), which identifies ambiguous observations in a deterministic process through simple consistency tests on the action values.

4.2.2 Memory-based Methods

Memory-based methods attempt to create a Markov representation through the use of past observations, very much like the information vector approach for Optimal Control. Unlike the attention-based techniques mentioned above, these methods do not assume that the external state can be identified among the immediate inputs. Instead, it considers the case when the use of past information is necessary in order to construct a Markovian representation of the process dynamics.

As for DP applied to partially observable processes, the fundamental problem here is the size of the information vector. In general, a suboptimal approach will be required in which the agent is equipped with either a) a feedback mechanism which allows past history to be encoded as an additional contextual information derived from past inputs or internal states; or b) an explicit limited memory of past observations (Figure 4.1).

Feedback architectures for the learning agent can be based on recurrent neural networks such as Elman networks (Elman, 1990), where a set of contextual features originated from previous hidden layer outputs is used as additional input.

A particular problem with feedback architectures is the dependency of the contextual information on both the states and the cost estimates. It may be difficult for the agent to identify if two given observations correspond to different states or just have different costs due to the noisy nature of reinforcement learning. A partial solution to this problem is the use of two separate structures: a recurrent one corresponding to a predictive model of the process and a feedforward one that generates the costs or action values using as inputs the current action and input, plus the contextual information provided by the predictive model. This separates the tasks of state estimation and cost estimation, but changes on the representation of contextual features may cause instability on cost learning (Whitehead and Lin, 1995).

[1] In section 4.2.3, we consider attentional methods that, combined with the use of a memory, can relax these restrictions.

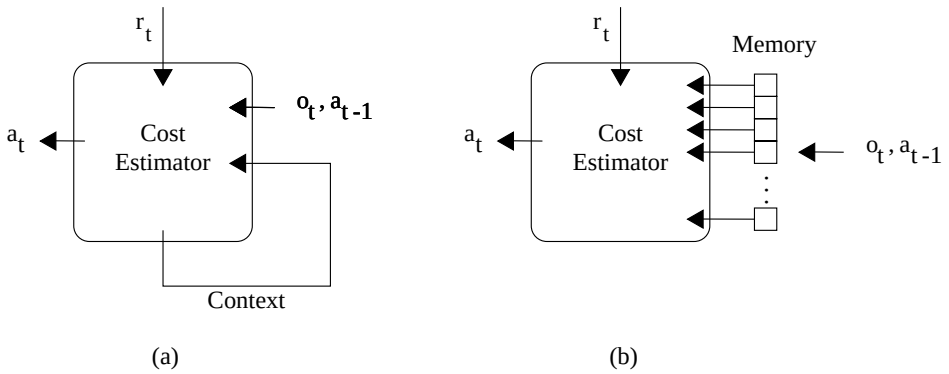

Figure 4.1: Memory-based methods. Case (a): use of feedback mechanism. Case (b): use of explicit memory.

An alternative approach is the use of an explicit memory that stores a limited set of past observations, in an attempt to create an 'approximate information vector'. This approach is particularly useful in tasks where there is strong temporal correlation among past observations, or when relevant information about the present state concentrates on a few past observations. The agent then acts on extended states — represented by these information vectors — in the same way that it would act if the states were available. However, an important drawback when information vectors are used is the size of the new extended state space. Experience generalisation strategies can be particularly important in this situation, as it is very unlikely that an agent can not only sufficiently visit every single state, but also have physical space to store all the possible history of observations.

A workaround for this problem is the use of a technique based on storage of only the instances that have actually been experimented by the agent. Each experience is defined as a tuple $\langle a_t, o_t, r_t \rangle$, composed by the chosen action a_t and consequents observation o_t and reward r_t. McCallum (McCallum, 1996b) proposed calculating action values for each visited instance by averaging expected future rewards associated with the K nearest experiences of the instance sequence, in an analogy with the K-nearest neighbor method for pattern recognition in geometric spaces. This method is very simple to implement, and gives good results in some simple partially observable problems.

4.2.3 Combining Attention and Memory

Combination of attention-based and memory-based methods in the hope that different techniques may complement each other is a feasible approach to the learning problem in partially observable processes.

The Utile Distinction approach (McCallum, 1996a) and the Perceptual Distinctions (PD) method (Chrisman, 1992) are representatives of this category. Unlike standard attention-based methods, these techniques do not assume that states can be disambiguated by any immediate observation. Unlike standard memory-based methods they do use active perception.

Both Utile Distinctions and PD borrow concepts from Hidden Markov Models (HMM) theory (Rabiner, 1989) in order to update belief states π (vectors storing internal state occupation probabilities) using cumulated experience. The number of internal states is a measure of how complex the process model is, and this number also defines

the size of the belief vector π_t. Action value updates are carried out simply by computing one step of the Q-learning equation for every model state $\mathbf{x}$ using the current belief vector:

$$Q_{t+1}(\mathbf{x}, a_t) = Q_t(\mathbf{x}, a_t) + \alpha_t \pi_t^{\mathbf{x}} [r(\mathbf{x}, a_t) + \gamma \hat{V}_t(\pi_{t+1}) - Q_t(\mathbf{x}, a_t)] \tag{4.7}$$

where $\pi_t^{\mathbf{x}}$ is the $\mathbf{x}$ component of the belief vector and $\hat{V}_t(\pi_{t+1}) = \min_a [Q_t(\pi_{t+1}, a)]$. The model is periodically tested using statistical significance tests, and if judged insufficient, new distinctions (*i.e.,* internal states) are added by splitting existing states. For PD, these tests evaluate the statewise predictive capability of the model. For Utile Distinctions, the tests evaluate the agent's ability to predict *reward*. However, it is not necessary that the feature selection mechanism perfectly discriminate states at the end (as in G learning), because the predictive model uses a memory of the past to facilitate the task in both cases.

Utile Distinctions was originally proposed as a method to make only memory distinctions (McCallum, 1992), but was later extended to deal with perceptual selection as a true attention-based method (McCallum, 1996a). In this form, it is possibly the most general implementation of a perceptually selective agent to date, as it both selects percepts and memory distinctions. It has been successfully tested in complex navigation tasks, but it also has a memory limitation (in spite of its distinction capability), high computational demands, and no guarantees whatsoever that the statistical state splitting test is suitable for any learning task (McCallum, 1996a).

Appendix A

Internet Sites on RL-related subjects

Some of these sites were mentioned along the text.

www-anw.cs.umass.edu/ rich/book/the-book.html Site for *Reinforcement Learning: An Introduction*, book by Andrew Barto and Richard Sutton.

world.std.com/ athenasc/ndpbook.html Site for book *Neurodynamic Programming*, by Dimitri Bertsekas and John Tsitsiklis.

web.cps.msu.edu/rlr/ RL repository at Michigan State University. Pointers to many publications, addresses of people doing research in RL, calls for papers, *etc.*

www.cs.cmu.edu/afs/cs.cmu.edu/project/reinforcement/web/index.html The 'Machine Learning and Friends' page at Carnegie-Mellon University. Abstracts of recent talks given at the CMU group in Machine Learning (a good source of information on current research trends), links to other sites and Conference announcements.

www.gmd.de/ml-archive/ Machine Learning archive at GMD, Germany. A comprehensive Machine Learning site, may contain information relevant to RL researchers.

envy.cs.umass.edu/ rich/ Richard Sutton's homepage. Includes some software and demo programs for RL algorithms.

www.cs.colorado.edu/ baveja/Demo.html A Java demo for solving the channel allocation problem using RL, by Satinder Singh.

www.cs.cmu.edu/afs/cs/project/theo-11/www/mccallum/ Source for Andrew McCallum's Reinforcement Learning Toolkit. Includes code for the U-Tree algorithm.

mlis.www.wkap.nl/mach/ Site of the Machine Learning Journal, an excellent source of information on outstanding research in RL.

Bibliography

Albus, J. S. (1971). A theory of cerebellar functions. *Mathematical Biosciences*, 10:25–61.

Barto, A. G., Sutton, R. S., and Anderson, C. W. (1983). Neuronlike elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man and Cybernetics*, 13:834–846.

Barto, A. G., Sutton, R. S., and Watkins, C. J. C. H. (1990). Learning and sequential decision making. In *Learning and Computational Neuroscience: Foundations of Adaptive Networks*. MIT Press.

Bellman, R. (1957). *Applied Dynamic Programming*. Princeton University Press, Princeton, New Jersey.

Bertsekas, D. P. (1995a). A counterexample to temporal differences learning. *Neural Computation*, 7:270–279.

Bertsekas, D. P. (1995b). *Dynamic Programming and Optimal Control*, volume 1. Athena Scientific, Belmont, Massachusetts.

Bertsekas, D. P. and Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming*. Athena Scientific, Belmont, Massachusetts.

Blake, A. and Yuille, A., editors (1992). *Active Vision*. MIT Press, Cambridge, Massachusetts.

Boyan, J. A. and Moore, A. W. (1995). Generalization in reinforcement learning: Safely approximating the value function. In Tesauro, G., Touretzky, D. S., and Leen, T. K., editors, *Advances in Neural Information Processing Systems 7*. MIT Press.

Brooks, R. A. (1991). Elephants don't play chess. In Maes, P., editor, *Designing Autonomous Agents*, pages 3–15. MIT Press.

Brooks, R. A. and Mataric, M. J. (1993). Real robots, real learning problems. In Connell, J. H. and Mahadevan, S., editors, *Robot Learning*, chapter 8, pages 193–213. Kluwer Academic Publishers.

Chapman, D. and Kaelbling, L. P. (1991). Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Procs. of the International Joint Conf. on Artificial Intelligence (IJCAI'91)*, pages 726–731.

Chrisman, L. (1992). Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Procs. of the 10th National Conf. on Artificial Intelligence*, pages 183–188.

Crites, R. H. (1996). *Large-scale Dynamic Optimization Using Teams of Reinforcement Learning Agents*. PhD thesis, University of Massachusetts Amherst.

del R. Millán, J. (1996). Rapid, safe and incremental learning of navigation strategies. *IEEE Transactions on Systems, Man and Cybernetics - Part B: Cybernetics*, 26(3):408–420.

Dorigo, M. and Colombetti, M. (1994). Robot shaping: Developing autonomous agents through learning. *Artificial Intelligence*, 71:321–370.

Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14:179–211.

Gat, E., Desai, R., Ivlev, R., Locj, J., and Miller, D. P. (1994). Behavior control for robotic exploration of planetary surfaces. *IEEE Transactions on Robotics and Automation*, 10(4):490–503.

Haykin, S. (1999). *Neural Networks: A Comprehensive Foundation*. Prentice-Hall, 2 edition.

Holland, J. H. and Reitman, J. S. (1978). Cognitive systems based on adaptive algorithms. In Waterman, D. A. and Hayes-Roth, F., editors, *Pattern-Directed Inference Systems*, pages 313–329. Academic Press.

Jaakkola, T., Jordan, M. I., and Singh, S. P. (1994). On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6(6):1185–1201.

Kappen, H. J., Nijman, M. J., and van Morsel, T. (1995). Learning active vision. In Fogelman-Soulié, F. and Gallinari, P., editors, *Procs. of the International Conf. on Artificial Neural Networks (ICANN'95)*. EC2 et Cie.

Kuipers, B. J. (1987). A qualitative approach to robot exploration and map learning. In *AAAI Workshop on Spatial Reasoning and Multi-Sensor Fusion*.

Lin, L.-J. (1991). Self-improving reactive agents: Case studies of reinforcement learning frameworks. In *Procs. of the First International Conf. on Simulation of Adaptive Behavior: from Animals to Animats*, pages 297–305.

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

Lin, L.-J. and Mitchell, T. M. (1992). Memory approaches to reinforcement learning in non-markovian domains. CMU-CS-92 138, Carnegie Mellon University, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

Mahadevan, S. and Connell, J. (1992). Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence*, 55:311–365.

Matarić, M. J. (1990). A distributed model for mobile robot environment learning and navigation. Master's thesis, Massachusetts Institute of Technology.

Matarić, M. J. (1991). A comparative analysis of reinforcement learning methods. Technical report, Massachusetts Institute of Technology.

McCallum, A. K. (1996a). *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis, University of Rochester.

McCallum, R. A. (1992). First results with utile distinction memory for reinforcement learning. Technical Report 446, The University of Rochester, Computer Science Department, The University of Rochester, Rochester, NY 14627.

McCallum, R. A. (1996b). Hidden state and reinforcement learning with instance-based state identification. *IEEE Transactions on Systems, Man and Cybernetics - Part B: Cybernetics*, 26(3):464–473.

McGarity, M., Sammut, C., and Clements, D. (1995). Controlling a steel mill with BOXES. In Furukawa, K., Michie, D., and Muggleton, S., editors, *Machine Intelligence 14*, pages 299–321. Oxford University Press.

Thomas Miller, III, W., Glanz, F. H., and Gordon Kraft, III, L. (1990). CMAC: An associative neural network alternative to backpropagation. *Proceedings of the IEEE*, 78(10):1561–1567.

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

Moore, A. W. and Atkeson, C. G. (1993). Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13:103–130.

Ogata, K. (1994). *Designing Linear Control Systems with MATLAB*. Prentice-Hall.

Peng, J. and Williams, R. J. (1996). Incremental multi-step q-learning. *Machine Learning*, 22:283–290.

Puterman, M. L. (1994). *Markovian Decision Problems*. John Wiley.

Rabiner, L. R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2).

Ribeiro, C. H. C. (1998). Embedding a priori knowledge in reinforcement learning. *Journal of Intelligent and Robotic Systems*, 21(1):51–71.

Ribeiro, C. H. C. and Szepesvári, C. (1996). Q-Learning combined with spreading: Convergence and results. In *Procs. of the ISRF-IEE International Conf. on Intelligent and Cognitive Systems (Neural Networks Symposium)*, pages 32–36.

Robbins, H. and Monro, S. (1951). A stochastic approximation method. *Annals of Mathematical Statistics*, 22:400–407.

Russell, S. J. and Norvig, P. (1995). *Artificial Intelligence: a modern approach*. Prentice-Hall.

Saerens, M. and Soquet, A. (1989). A neural controller. In *Procs. of the 1st IEE International Conf. on Artificial Neural Networks*, pages 211–215, London. The Institution of Electrical Engineers.

Sammut, C. A. (1994). Recent progress with BOXES. In Furukawa, K., Muggleton, S., and Michie, D., editors, *Machine Intelligence 13*. The Clarendon Press, Oxford.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, 3:210–229.

Singh, S. P. (1994). *Learning to Solve Markovian Decision Processes*. PhD thesis, University of Massachusetts.

Singh, S. P. and Bertsekas, D. (1997). Reinforcement learning for dynamic channel allocation in cellular telephone systems. In Mozer, M. C., Jordan, M. I., and Petsche, T., editors, *Advances in Neural Information Processing Systems 9*. MIT Press.

Singh, S. P. and Dayan, P. (1996). Analytical mean squared error curves for temporal difference learning. *Machine Learning*. In press.

Singh, S. P., Jaakkola, T., and Jordan, M. I. (1995). Reinforcement learning with soft state aggregation. In Tesauro, G., Touretzky, D. S., and Leen, T. K., editors, *Advances in Neural Information Processing Systems 7*, pages 361–368. MIT Press.

Singh, S. P. and Sutton, R. S. (1996). Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22:123–158.

Striebel, C. T. (1965). Sufficient statistics in the optimal control of stochastic systems. *Journal of Math. Analysis and Applications*, 12:576–592.

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

Sutton, R. S. (1990). Integrated architectures for learning, planning and reacting based on approximating dynamic programming. In *Procs. of the 7th International Conf. on Machine Learning*, pages 216–224.

Sutton, R. S. (1996). Generalization in reinforcement learning: Succesful examples using sparse coarse coding. In Touretzky, D. S., Mozer, M. C., and Hasselmo, M. E., editors, *Advances in Neural Information Processing Systems 8*, pages 1038–1044. MIT Press.

Sutton, R. S. and Barto, A. G. (1990). Time-derivative models of pavlovian reinforcement. In *Learning and Computational Neuroscience: Foundations of Adaptive Networks*. MIT Press.

Szepesvári, C. (1997). *Static and Dynamic Aspects of Optimal Sequential Decision Making*. PhD thesis, József Attila University, Szeged, Hungary.

Szepesvári, C. and Littman, M. L. (1996). Generalized markov decision processes: Dynamic-programming and reinforcement-learning algorithms. CS-96-11, Brown University, Department of Computer Science, Brown University, Providence, Rhode Island 02912.

Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning*, 8:257–277.

Tesauro, G. (1995). Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–67.

Tsitsiklis, J. N. and Roy, B. V. (1996). Feature-based methods for large scale dynamic programming. *Machine Learning*, 22:59–94.

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

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

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

Whitehead, S. D. and Lin, L.-J. (1995). Reinforcement learning of non-markov decision processes. *Artificial Intelligence*, 73:271–306.

Widrow, B., Gupta, N. K., and Maitra, S. (1973). Punish/reward: Learning with a critic in adaptive threshold systems. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-3(5):455–465.

Widrow, B. and Hoff, M. E. (1960). Adaptive switching circuits. In *IRE WESCON Convention Record*, volume 4, pages 96–104, New York.

Widrow, B. and Smith, F. W. (1963). Pattern recognizing control systems. In *1963 Computer Info. Sci. (COINS) Symposium*, pages 288–317, Washington, DC.

Witten, I. H. (1977). An adaptive optimal controller for discrete-time markov environments. *Information and Control*, 38:286–295.