Chapter 2

Reinforcement Learning

2.1. Introduction

In Chapter 1, we presented planning methods in which the agent knows the transition and reward functions of the Markov decision problem it faces. In this chapter, we present *reinforcement learning* methods, where the transition and reward functions are not known in advance.

EXAMPLE.– Let us consider again the case of a car used in the introduction of the previous chapter (see Section 1.1). If one must look after a type of car never met before and does not possess the corresponding manual, one cannot directly model the problem as an MDP. One must first determine the probability of each breakdown, the cost of each repair operation and so on. In such a case, reinforcement learning is a way to determine through incremental experience the best way to look after the car by trial and error, eventually without even determining explicitly all probabilities and costs, just relying on a locally updated value of each action in each situation.

2.1.1. *Historical Overview*

Our presentation strongly relies on Sutton and Barto's book [SUT 98]. But before presenting the main concepts of reinforcement learning, we give a brief overview of the successive stages of research that led to the current formal understanding of the domain from the computer science viewpoint.

Chapter written by Olivier SIGAUD and Frédérick GARCIA.

Most reinforcement learning methods rely on simple principles coming from the study of animal or human cognition, such as the increased tendency to perform an action in a context if its consequences are generally positive in that context.

The first stages of computer science research that led to the reinforcement learning framework are dating back to 1960. In 1961, Michie [MIC 61] described a system playing tic-tac-toe by trial and error. Then Michie and Chambers [MIC 68] designed in 1968 a software maintaining an inverted pendulum's balance. In parallel, Samuel [SAM 59] presented a program learning to play checkers using a temporal difference principle. Both components, trial and error exploration and learning action sequences from temporal difference principles are the basis of all subsequent reinforcement learning systems.

The union of both principles was achieved by Klopf [KLO 72, KLO 75]. Following his work, Sutton and Barto implemented in 1981 [SUT 81] a linear perceptron whose update formula directly derives from the Rescorla-Wagner equation [RES 72], coming from experimental psychology. The Rescorla-Wagner equation can now be seen as the $TD(0)$ equation (see Section 2.5.1, page 63) approximated by a linear perceptron. This historical background explains the name *neuro-dynamic programming* used in the early stages of the field [BER 96].

In 1983, Barto, Sutton and Anderson [BAR 83] proposed AHC-LEARNING[1] considered as the first true reinforcement learning algorithm. Quite interestingly, AHC-LEARNING is an actor-critic approach that is now at the heart of the recent dialog between computational modeling and neurophysiological understanding of reinforcement learning in animals.

Finally, the mathematical formalisation of reinforcement learning as we know it today dates back to 1988 when Sutton [SUT 88] and then Watkins [WAT 89] linked their work to the optimal control framework proposed by Bellman in 1957, through the notion of MDP [BER 95].

2.2. Reinforcement Learning: a Global View

Reinforcement learning stands at the intersection between the field of dynamic programming presented in the previous chapter and the field of machine learning. As a result, there are two ways to introduce the domain, either by explaining the conceptual differences with the dynamic programming context, or by contrasting reinforcement learning with other machine learning paradigms. We take the former

1. *Adaptive Heuristic Critic learning*

standpoint in what follows and the latter in Section 2.2.2. Then we introduce the *exploration/exploitation trade-off* and a standard estimation method that are central to reinforcement learning.

2.2.1. *Reinforcement Learning as Approximate Dynamic Programming*

As we have seen in the previous chapter, the value function of a state typically reflects an estimate of the cumulated reward an agent might expect from being in this state given its current behaviour. This notion is closely related to the notion of evaluation function in game theory. It distinguishes reinforcement learning approaches from all other simulation-based methods such as evolutionary algorithms that can also build optimal policies but without using the temporal structure of the underlying sequential decision problems.

Most reinforcement learning methods presented in this chapter are closely related to the dynamic programming algorithms presented in the previous chapter. Indeed, reinforcement learning can be seen as an extension of dynamic programming to the case where the dynamics of the problem is not known in advance.

Reinforcement learning methods are said to be model-free or model-based depending on whether they build a model of the transition and reward functions $p(s' \mid s, a)$ and $r(s, a)$ of the underlying MDP. As a matter of fact, dynamic programming methods can be seen as a special case of indirect (aka model-based) reinforcement learning methods where the model is perfect. When the model is unknown, model-based methods must build it on-line. In the discrete case studied in this chapter, this is done through a simple cumulative approach based on the maximum likelihood principle. In parallel, dynamic programming methods can be applied to the increasingly accurate model.

Direct (aka model-free) methods do not build a model: the hidden $p(s' \mid s, a)$ and $r(s, a)$ parameters are not estimated, but the value function V is updated locally while experimenting. This approach is advantageous in terms of memory use and, historically, reinforcement learning was initially restricted to direct methods. The central idea of direct reinforcement learning consists in improving a policy *locally* after each interaction with the environment. The update is local, thus it does not require a global evaluation of the policy. In practice, however, most direct reinforcement learning algorithms do not work directly on the policy, but iterate on an approximate value function as defined in the previous chapter.

All methods presented in this chapter deal with the case where the transition and reward functions of the MDP are unknown and an optimal value function V^* or a function Q^* representing the value of each action in each state will be approximated through experience, either on a model-based or a model-free basis.

2.2.2. *Temporal, Non-Supervised and Trial-and-Error based Learning*

Reinforcement learning can be distinguished from other forms of learning based on the following characteristics:

– Reinforcement learning deals with temporal sequences. In contrast with one-step supervised or non-supervised learning problems where the order in which the examples are presented is not relevant, the choice of an action at a given time step will have consequences on the examples that are received at the subsequent time steps. This is particularly critical in the context of *delayed reward* problems where a reward may be received far after the important choices have been made.

– In contrast with supervised learning, the environment does not tell the agent what would be the best possible action. Instead, the agent may just receive a scalar reward representing the value of its action and it must *explore* the possible alternative actions to determine whether its action was the best or not.

– Thus, in order to determine the best possible action in any situation, the agent must try a lot of actions, through a trial-and-error process that implies some exploration, giving rise to the *exploration/exploitation trade-off* that is central to reinforcement learning.

2.2.3. *Exploration versus Exploitation*

Exploitation consists in doing again actions which have proved fruitful in the past, whereas exploration consists in trying new actions, looking for a larger cumulated reward, but eventually leading to a worse performance. In theory, as long as the agent has not explored all possible actions in all situations, it cannot be sure that the best policy it knows is optimal. The problem is even more accurate when the environment is stochastic, leading to the necessity to try each action in each situation several times to get a reliable estimate of its average value. As a result, all convergence proofs for reinforcement learning algorithms assume that each transition will be experienced infinitely often [WAT 92]. In practice, however, one must do with a partial exploration. Dealing with the exploration/exploitation trade-off consists in determining how the agent should explore to get as fast as possible a policy that is optimal or close enough to the optimum.

The agent may explore states as well as actions. With respect to states, the most natural choice consists in just following the dynamics of the agent-environment system, using the state resulting from the previous action as state of the next learning step. This way, exploration is focused on the relevant part of the state space that will be covered in practice by the agent. On-line planning methods such as RTDP rely on this idea (see Chapter 6). Furthermore, there are many practical settings where doing otherwise is extremely difficult. In robotics, for instance, it may be hard not to follow the natural dynamics of the robot. Note however that in settings where there is a state

or set of states where the system stays once it enters it, it is generally necessary to reinitialize the system somewhere else to keep exploring.

With respect to actions, sampling the action uniformly in the action space at each iteration satisfies the convergence criterion, ensuring a maximal exploration, but such a choice is inefficient for two reasons. First, the value of the best action in each state is updated as often as the value of the worst action, which results in robust, but slow learning. Second, the cumulated reward along the experiment is just the average performance over all possible policies, which may be inadequate when learning is performed on the target system.

On the opposite, choosing at each iteration the greedy action with respect to the current known values, i.e. performing a greedy policy is not satisfactory either, because it generally converges to a suboptimal policy or may even diverge.

Thus reinforcement learning algorithms are based on a trade-off between full exploration and full exploitation which generally consists in performing the greedy action most of the time and an exploratory action from time to time. In this setting, finding the optimal way of tuning the rate of exploratory actions along an experiment is still an open problem.

Methods to deal with the exploration/exploitation trade-off can be classified into two categories: undirected methods and directed methods [THR 92].

Undirected methods use few information from the learning experiments beyond the value function itself. For instance, one may [BER 96]:

– follow the greedy policy along N_1 iterations, then perform random exploration along N_2 iterations;

– follow at each iteration the greedy policy with probability $1-\epsilon$ or a random policy with probability ϵ, with $\epsilon \in [0,1]$; these methods are called *ϵ-greedy*;

– draw an action according to a Boltzmann distribution, i.e the probability of drawing action a is

$$p_T(a) = \frac{\exp(-\frac{Q_n(s_n,a)}{T})}{\sum_{a'} \exp(-\frac{Q_n(s_n,a')}{T})}$$
$$\text{with } \lim_{n\to\infty} T = 0$$

where $Q_n(s,a)$ is the action-value function of performing action a in state s; these methods are called *softmax*. The roulette wheel selection method is a particular softmax method in which T is constant instead of decreasing.

These diverse exploration functions call upon parameters (N_1, N_2, ϵ and T) controlling the exploration rate. The useful cases are $N_2 > 0$, $T < +\infty$ and $\epsilon < 1$, which experimentally ensure fast convergence.

By contrast, directed methods use specific exploration heuristics based on the information available from learning. Most such methods boil down to adding some exploration bonus to $Q(s,a)$ [MEU 96]. This bonus can be local such as in the interval estimation method [KAE 93], or propagated from state to state during learning [MEU 99]. Simple definitions of this exploration bonus can lead to efficient methods:

– in the *recency-based method*, the bonus is $\varepsilon\sqrt{\delta n_{sa}}$ where δn_{sa} is the number of iterations since the last execution of action a in state s, and where $\varepsilon \in [0, 1[$;

– in the *uncertainty estimation method*, the bonus is $\frac{c}{n_{sa}}$, where c is a constant and n_{sa} is the number of times action a was chosen in state s.

These different exploration methods can be used in the context of any reinforcement learning algorithm. Indeed, in all the temporal difference learning algorithms presented below, convergence towards optimal values is guaranteed provided that the Markov assumption is fulfilled and that each state is visited an infinite number of times.

2.2.4. *General Preliminaries on Estimation Methods*

Before presenting general temporal difference methods in the next section, we will present Monte Carlo methods as a specific instance of these methods. For the sake of clarity, we will consider a policy π such that the corresponding Markov chain $p(s_{t+1} \mid s_t) = p(s_{t+1} \mid s_t, \pi(s_t))$ reach from any initial state an (absorbing) terminal state T with a null reward. We consider the total reward criterion and we try to approximate $V(s) = E(\sum_{t=0}^{\infty} R_t \mid s_0 = s)$.

A simple way of performing this estimation consists in using the average cumulated reward over different trajectories obtained by following a policy π. If $R_k(s)$ is the expected utility in state s along trajectory k, then an estimation of the value function V in s based on the average after $k+1$ trajectories is:

$$\forall s \in S, V_{k+1}(s) = \frac{R_1(s) + R_2(s) + ... + R_k(s) + R_{k+1}(s)}{k+1}. \tag{2.1}$$

To avoid storing all the rewards, this computation can be reformulated in an incremental way:

$$\forall s \in S, V_{k+1}(s) = V_k(s) + \frac{1}{k+1}[R_{k+1}(s) - V_k(s)]. \tag{2.2}$$

To get $V_{k+1}(s)$, one just needs to store $V_k(s)$ and k. One can even avoid storing k, using an even more generic formula:

$$\forall s \in S, V_{k+1}(s) = V_k(s) + \alpha[R_{k+1}(s) - V_k(s)] \tag{2.3}$$

where α is positive and should decrease along time, and we get:

$$\lim_{k \to \infty} V_k(s) = V^\pi(s) \quad (2.4)$$

This incremental estimation method is at the heart of temporal difference methods, but is also present in Monte Carlo.

2.3. Monte Carlo Methods

The Monte Carlo approach consists in performing a large number of trajectories from all states s in S, and estimating $V(s)$ as an average of the cumulated rewards observed along these trajectories. In each trial, the agent records its transitions and rewards, and updates the estimates of the value of the encountered states according to a discounted reward scheme. The value of each state then converges to $V^\pi(s)$ for each s if the agent follows policy π.

Thus the main feature of Monte Carlo methods lies in the incremental estimation of the value of a state given a series of cumulated reward values resulting from running a set of trajectories. The estimation method itself is the one presented in Section 2.2.4.

More formally, let $(s_0, s_1, \ldots, s_N)$ be a trajectory consistent with the policy π and the unknown transition function $p()$, and let $(r_1, r_2, \ldots, r_N)$ be the rewards observed along this trajectory.

In the Monte Carlo method, the N values $V(s_t)$, $t = 0, \ldots, N-1$ are updated according to:

$$V(s_t) \leftarrow V(s_t) + \alpha(s_t)(r_{t+1} + r_{t+2} + \cdots + r_N - V(s_t)) \quad (2.5)$$

with the learning rates $\alpha(s_t)$ converging to 0 along the iterations. Then V converges almost surely towards V^π under very general assumptions [BER 96].

This method is said "every-visit" because the value of a state can be updated several times along the same trajectory. The error terms corresponding to all these updates are not independent, giving rise to a bias in the estimation of the V function along a finite number of trajectories [BER 96, page 190]. A simple solution to this bias problem consists in only updating $V(s)$ on the first visit of s in each trajectory. This "first-visit" method is not biased. Experimentally, the average quadratic error of the *first-visit* method tends to be lower than that of the *every-visit* method [SIN 96].

2.4. From Monte Carlo to Temporal Difference Methods

The standard Monte Carlo methods above update the value function at the end of each trajectory. They can be improved so as to perform updates after every transition. The update rule (2.5) can be rewritten in the following way, giving rise to a more incremental method:[2]

$$\begin{aligned} V(s_t) \leftarrow V(s_t) + \alpha(s_t) \quad & \Big((r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \\ & +(r_{t+2} + \gamma V(s_{t+2}) - V(s_{t+1})) \\ & + \cdots \\ & +(r_N + \gamma V(s_N) - V(s_{N-1}))\Big) \end{aligned}$$

or

$$V(s_t) \leftarrow V(s_t) + \alpha(s_t)(\delta_t + \delta_{t+1} + \cdots + \delta_{N-1}) \tag{2.6}$$

by defining the temporal difference error δ_t by

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t), \quad t = 0, \ldots, N-1.$$

The error δ_t can be interpreted in each state as a measure of the difference between the current estimation $V(s_t)$ and the corrected estimation $r_{t+1} + V(s_{t+1})$. It can be computed as soon as the transition (s_t, r_{t+1}, s_{t+1}) has been observed, giving rise to an "on-line" version of the update rule (2.6). Thus one can start updating V without waiting for the end of the trajectory.

$$V(s_l) \leftarrow V(s_l) + \alpha(s_l)\delta_t, \quad l = 0, \ldots, t \tag{2.7}$$

Once again, if a trajectory can visit the same state several times, the on-line version can differ from the original version (2.6). However, it still converges almost surely and the first-visit method still seems more efficient in practice [SIN 96].

Thus, whereas standard Monte Carlo methods require that each trajectory is finished to perform updates, the on-line version presented just above can be said incremental. We can now turn to temporal difference methods that combine the incrementality property of dynamic programming and the experience based update mechanism of Monte Carlo methods.

2. γ is 1, it is introduced in the equations to highlight the relationship with the temporal difference error used in temporal difference methods. Furthermore, we use $V(s_N) = V(T) = 0$

2.5. Temporal Difference Methods

So far, we have seen two classes of behaviour optimisation algorithms:

– Dynamic programming algorithms apply when the agent knows the transition and reward functions. They perform their updates locally, which results in the possibility to act without waiting for the end of all iterations. However, they require a perfect knowledge of the underlying MDP functions.

– By contrast, Monte Carlo methods do not require any knowledge of the transition and reward functions, but they are not local.

Temporal difference methods combine properties of both previous methods. They rely on an incremental estimation of the value or action-value functions. Like Monte Carlo methods, they perform this estimation based on the experience of the agent and can do without a model of the underlying MDP. However, they combine this estimation using local estimation propagation mechanisms coming from dynamic programming, resulting in their incremental properties.

Thus temporal difference methods, which are at the heart of most reinforcement learning algorithms, are characterised by this combination of estimation methods with local updates incremental properties.

2.5.1. *The* TD(0) *Algorithm*

The different criteria that can be used to determine the cumulated reward along a trajectory from local rewards have been presented in the previous chapter. Among these criteria, we will focus in this section on the discounted reward that leads to the most classical studies and proofs.

The basic temporal difference algorithm is TD. We note it here $TD(0)$ for reasons that will get clear in the section dedicated to eligibility traces. This algorithm relies on a comparison between the actually received reward and the reward expected from the previous estimations.

If the estimated values $V(s_t)$ and $V(s_{t+1})$ in states s_t and s_{t+1} were exact, we would have:

$$V(s_t) = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... \quad (2.8)$$

$$V(s_{t+1}) = r_{t+2} + \gamma r_{t+3} + \gamma^2 r_{t+4} + ... \quad (2.9)$$

Thus we would have:

$$V(s_t) = r_{t+1} + \gamma V(s_{t+1}). \quad (2.10)$$

As stated before in Section 2.4, the temporal difference error $\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$ measures the error between the current estimation $V(s_t)$ and the corrected estimation $r_{t+1} + V(s_{t+1})$. The temporal difference method consists in correcting this error little by little by modifying $V()$ according to a Widrow-Hoff equation, often used in neural networks:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] = V(s_t) + \alpha\delta_t. \tag{2.11}$$

This update equation immediately reveals the connection between temporal difference methods, Monte Carlo methods and dynamic programming. Indeed, it combines two features:

– as in dynamic programming algorithms, the estimate of $V(s_t)$ is updated as a function of the estimate of $V(s_{t+1})$. Thus the estimation is propagated to the current state from the successor states;

– as in Monte Carlo methods, each of these values results from an estimation based on the experience of the agent along its interactions with its environment.

One can thus see that temporal difference methods and, in particular, TD(0), are based on two coupled convergence processes, the first one estimating the immediate reward in each state and the second one approximating the value function resulting from these estimates by propagating them along transitions.

In the context of TD(0), the updates are local each time the agent performs a transition in its environment, relying on an information limited to (s_t, r_{t+1}, s_{t+1}). The convergence of TD(0) was proven by Dayan and Sejnowski [DAY 94].

However, one must note that knowing the exact value of all states is not enough to determine what to do. If the agent does not know which action results in reaching any particular state, i.e. if the agent does not have a model of the transition function, knowing V does not help it determine its policy. This is why similar algorithms based on state-action pairs and computing the action-value function Q were developed, as we will show below.

2.5.2. *The* SARSA *Algorithm*

As we just explained, finding V through $V = LV$ does not result in a policy if the model of transitions in unknown. To solve this problem, Watkins [WAT 89] introduced the action-value function Q, whose knowledge is similar to the knowledge of V when p is known.

DEFINITION 2.1.– *Action-value function Q*
The action-value function of a fixed policy π whose value function is V^π is:

$$\forall s \in S, a \in A \quad Q^\pi(s,a) = r(s,a) + \gamma \sum_{s'} p(s' \mid s,a) V^\pi(s').$$

The value of $Q^\pi(s,a)$ is interpreted as the expected value when starting from s, executing a and then following the policy π afterwards. We have $V^\pi(x) = Q^\pi(x, \pi(x))$ and the corresponding Bellman equation is:

$$\forall s \in S, a \in A \quad Q^*(s,a) = r(s,a) + \gamma \sum_{s'} p(s' \mid s,a) \max_b Q^*(s',b).$$

Then we have

$$\forall s \in S \quad V^*(s) = \max_a Q^*(s,a) \text{ and}$$
$$\pi^*(s) = \operatorname*{argmax}_a Q^*(s,a).$$

The SARSA algorithm is similar to TD(0) in all respects but the fact that it works on state-action pairs rather than on states. Its update equation is the following:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)]. \quad (2.12)$$

The information necessary to perform such an update is $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$, hence the name of the algorithm.

The SARSA algorithm suffers from one conceptual drawback: performing the updates as stated above implies to know in advance what will be the next action a_{t+1} for any possible next state s_{t+1}. As a result, the learning process is tightly coupled to the current policy (the algorithm is said "on-policy") and this complicates the exploration process. As a result, proving the convergence of SARSA was more difficult than proving the convergence of "off-policy" algorithms such as Q-learning, presented below, thus the corresponding convergence proof was published much later [SIN 00].

Note however that empirical studies often demonstrate the better performance of SARSA compared to Q-learning.

2.5.3. *The Q-learning Algorithm*

The Q-learning algorithm can be seen as a simplification of the SARSA algorithm, given that it is no more necessary to determine the action at the next step to compute updates. Its update equation is the following:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]. \qquad (2.13)$$

Algorithm 2.1: Q-learning

```
/* α_t is the learning rate                                  */
Initialise(Q_0)
for t ← 0 to T_tot − 1 do
    s_t ← ChooseState
    a_t ← ChooseAction
    (s_{t+1}, r_{t+1}) ← Simulate(s_t, a_t)
    { update Q_t:}
    begin
        Q_{t+1} ← Q_t
        δ_t ← r_{t+1} + γ max_b Q_t(s_{t+1}, b) − Q_t(s_t, a_t)
        Q_{t+1}(s_t, a_t) ← Q_t(s_t, a_t) + α_t(s_t, a_t)δ_t
    end
return Q_{T_tot}
```

The main difference between SARSA and Q-learning lies in the definition of the error term. The $Q(s_{t+1}, a_{t+1})$ term in Equation 2.12 is replaced by $max_a Q(s_{t+1}, a)$ in Equation 2.13. It would be equivalent in the context of a greedy policy since we would have $a_{t+1} = \arg\max_a Q(s_{t+1}, a)$. But, given the necessity to deal with the exploration/exploitation trade-off, one can perform a non greedy action choice while still using the max term in the update rule. Thus SARSA performs updates in function of the actually chosen actions whereas Q-learning performs updates in function of the optimal actions irrespective of the actions performed, which is simpler.

This greater simplicity resulted both in earlier proofs of its convergence [WAT 92] and in the fact that it has been the most used algorithm over years, despite its eventual lower performance.

The Q-learning algorithm is shown in Algorithm 2.1. Updates are based on instantaneously available information. In this algorithm, the T_{tot} parameter corresponds to the number of iterations. There is here one learning rate $\alpha_t(s, a)$ for each state-action pair, it decreases at each visit of the corresponding pair. The `Simulate` function returns a new state and the corresponding reward according to the dynamics of the

system. The choice of the current state and of the executed action is performed by functions `ChooseState` and `ChooseAction` and will be discussed hereafter. The `Initialise` function initialises the Q function with Q_0, which is often initialised will null values, whereas more adequate choices can highly improve the performance.

The Q-learning algorithm can also be seen as a stochastic formulation of the *value iteration* algorithm presented in the previous chapter. Indeed, *value iteration* can be expressed in terms of action value function:

$$\begin{array}{rl} & V_{n+1}(s) = \max_{a \in A} \overbrace{\left\{ r(s,a) + \gamma \sum_{s' \in S} p(s' \mid s,a) V_n(s') \right\}}^{Q_n(s,a)} \\ \Rightarrow & Q_{n+1}(s,a) = r(s,a) + \gamma \sum_{s' \in S} p(s' \mid s,a) V_{n+1}(s') \\ \Rightarrow & Q_{n+1}(s,a) = r(s,a) + \gamma \sum_{s' \in S} p(s' \mid s,a) \max_{a' \in A} Q_n(s',a') \end{array}$$

The Q-learning algorithm is obtained by replacing $r(s,a) + \sum_{s'} p(s' \mid s,a) \max_{a' \in A} Q(s',a')$ by its simplest unbiased estimator built from the current transition $r_{t+1} + \max_{a' \in A} Q(s_{t+1},a')$.

The convergence of this algorithm is proved [WAT 89, JAA 94] (Q_n converges almost surely to Q^*) under the following assumptions:

– S and A are finite,

– each (s,a) pair is visited an infinite number of times,

– $\sum_n \alpha_n(s,a) = \infty$ and $\sum_n \alpha_n^2(s,a) < \infty$,

– $\gamma < 1$ or, if $\gamma = 1$, there exists an absorbing state with null reward for any policy.

The almost sure convergence means that, $\forall s,a$, the sequence of $Q_n(s,a)$ converges to $Q^*(s,a)$ with a probability equal to 1. In practice, the sequence $\alpha_n(s,a)$ is often defined as $\alpha_n(s,a) = \frac{1}{n_{sa}}$.

2.5.4. *The TD(λ), Sarsa(λ) and Q(λ) Algorithms*

The TD(0), SARSA and Q-learning algorithms only perform one update per time step, in the state that the agent is visiting. As shown in Figure 2.1, this update process is particularly slow. Indeed, an agent deprived of any information on the structure of the value function needs at least n trials to propagate the immediate reward of a state to another state that is n transitions away. Before this propagation is achieved, if the initial values are null, the agent performs a random walk in the state space, which means that it needs an exponential number of steps in function of n before reaching the reward "trail".

Figure 2.1. *Q-learning: first and second trial. One can see that, all values being initially null, the propagation of non-null values does not start until the agent finds the reward source for the first time, and only progresses once per trial.*

A first naive way to solve the problem consists in using a memory of the trajectory and to propagate all the information backwards along the performed transitions each time a reward is reached. Such a memory of performed transitions is called an "eligibility trace".

Based on this idea, Sutton and Barto [SUT 98] proposed a class of algorithms called "TD(λ)" that generalise TD(0) to the case where the agent uses a memory of transitions. Later, SARSA and Q-learning have also been generalised into SARSA(λ) and Q(λ), by two different ways and two different authors for the latter [WAT 92, PEN 96].

These algorithms are more efficient than their standard counterpart, but they require more memory.

A problem with the naive approach above is that the required memory grows with the length of trajectories, which is obviously not feasible in the infinite horizon context.

In TD(λ), SARSA(λ) and Q(λ), a more sophisticated approach that addresses the infinite horizon case is used.

We will discuss in Section 2.6.1, page 76, another solution that performs several updates at each time step, in the indirect (or model-based) reinforcement learning framework. Let us start by describing the TD(λ), SARSA(λ) and $Q(\lambda)$ algorithms in more details.

2.5.5. *Eligibility Traces and TD(λ)*

The originality of the TD(λ) method lies in the proposal of a compromise between Equations 2.6 and 2.7. Let $\lambda \in [0, 1]$ be a parameter. With the same notations as

before, the TD(λ) algorithm defined by Sutton [SUT 88] is:

$$V(s_t) \leftarrow V(s_t) + \alpha(s_t) \sum_{m=t}^{N-1} \lambda^{m-t} \delta_m, \quad t = 0, \ldots, N-1 \tag{2.14}$$

The role of λ can be understood by rewriting Equation 2.14 as follows:

$$V(s_t) \leftarrow V(s_t) + \alpha(s_t)(z_t^\lambda - V(s_t)).$$

Then we have

$$\begin{aligned} z_t^\lambda &= V(s_t) + \sum_{m=t}^{N-1} \lambda^{m-t} \delta_m \\ &= V(s_t) + \delta_t + \lambda \sum_{m=t+1}^{N-1} \lambda^{m-t-1} \delta_m \\ &= V(s_t) + \delta_t + \lambda(z_{t+1}^\lambda - V(s_{t+1})) \\ &= V(s_t) + r_{t+1} + V(s_{t+1}) - V(s_t) + \lambda(z_{t+1}^\lambda - V(s_{t+1})) \\ &= r_{t+1} + (\lambda z_{t+1}^\lambda + (1-\lambda)V(s_{t+1})) \end{aligned}$$

In the $\lambda = 0$ case, this is equivalent to using a one step horizon, as in dynamic programming. Thus this is TD(0).

If $\lambda = 1$, Equation 2.14 can be rewritten

$$V(s_t) \leftarrow V(s_t) + \alpha(s_t) \sum_{m=t}^{N-1} \delta_m, \quad t = 0, \ldots, N-1,$$

which is exactly Equation 2.5 in Monte Carlo methods.

For any λ, both *first-visit* and *every-visit* approaches can be considered. An *on-line* version of the TD(λ) learning algorithm described by Equation 2.14 is possible:

$$V(s_l) \leftarrow V(s_l) + \alpha(s_l)\lambda^{t-l}\delta_t, \quad l = 0, \ldots, t \tag{2.15}$$

as soon as the transition (s_t, r_{t+1}, s_{t+1}) is performed and the δ_t error is computed.

Applying TD(λ) to evaluate a policy π according to the discounted criterion implies some modifications to the standard Algorithms 2.14 or 2.15.

In the $\gamma = 1$ case, the update rule is the following:

$$V(s_t) \leftarrow V(s_t) + \alpha(s_t) \sum_{m=t}^{\infty} (\gamma\lambda)^{m-t} \delta_m. \quad (2.16)$$

It is then clear that an off-line algorithm to compute V is inadequate in the absence of an absorbing terminal state since the trajectory is potentially infinite. The on-line version of (2.16) is then defined as:

$$V(s) \leftarrow V(s) + \alpha(s) z_t(s) \delta_t, \quad \forall s \in S, \quad (2.17)$$

as soon as the tth transition (s_t, r_{t+1}, s_{t+1}) is performed and the δ_t error is computed. The eligibility trace $z_t(s)$ is defined as follows:

DEFINITION 2.2.– *Cumulative eligibility trace*

$$\begin{array}{ll} z_0(s) & = 0, \quad \forall s \in S, \\ z_n(s) & = \begin{cases} \gamma\lambda z_{n-1}(s) & \text{if } s \neq s_n, \\ \gamma\lambda z_{n-1}(s) + 1 & \text{if } s = s_n. \end{cases} \end{array}$$

The eligibility coefficient increases at each visit of the corresponding state and exponentially decreases during the other iterations until a new visit of that state (see Figure 2.2).

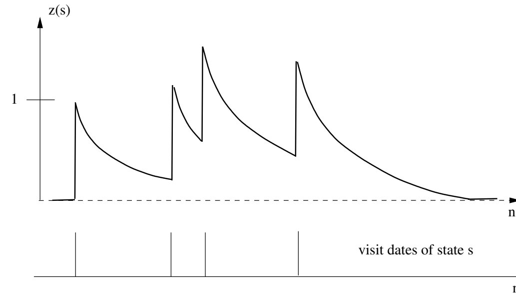

Figure 2.2. *Cumulative eligibility trace: at each visit, one adds 1 to the previous value, thus this value can get over 1*

In some cases, a slightly different definition of $z_n(s)$ seems to lead to a faster convergence of V.

DEFINITION 2.3.– *Eligibility trace with reinitialisation*

$$\begin{array}{ll} z_0(s) & = 0 \quad \forall s \in S, \\ z_n(s) & = \begin{cases} \gamma\lambda z_{n-1}(s) & \text{if } s \neq s_n, \\ 1 & \text{if } s = s_n. \end{cases} \end{array}$$

Thus the value of the trace is bounded by 1, as shown in Figure 2.3.

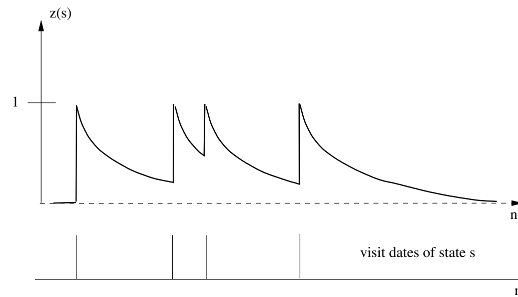

Figure 2.3. *Eligibility trace with reinitialisation: the value is set to 1 at each visit*

The almost sure convergence of the TD(λ) algorithm was shown for any λ, both with on-line and off-line approaches, under classical assumptions of infinite visits of each state $s \in S$ and convergence of α towards 0 at each iteration n, such that $\sum_n \alpha_n(s) = \infty$ and $\sum_n \alpha_n^2(s) < \infty$ [JAA 94, BER 96].

The effect of the λ value is still poorly understood and tuning optimally its value for a given problem is still an open empirical problem.

A direct implementation of TD(λ) based on eligibility traces is not efficient as soon as the state space S becomes large. A first approximate solution [SUT 98] consists in setting to 0 the value of all traces $z_n(s) < \varepsilon$, thus in stopping to maintain traces that have not been visited since more than $\frac{\log(\varepsilon)}{\log(\gamma\lambda)}$ transitions.

Another approximate method known as "truncated temporal differences", or TTD(λ) [CIC 95], is equivalent to storing a window of size m memorising the m last visited state and updating from this window at each iteration n the value of the state visited in iteration $(n - m)$.

2.5.6. *From TD(λ) to Sarsa(λ)*

TD(λ) can be applied in a reinforcement learning context to learn an optimal policy. To do so, one can couple TD(λ) with an algorithm storing a sequence of policies π_t, as will become clear when presenting actor-critic approaches (see Chapter 5).

However, Q-learning directly integrates the temporal difference error idea. With the update rule of Q-learning:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha_t\{r_{t+1} + \gamma V_t(s_{t+1}) - Q_t(s_t, a_t)\}$$

for transition $(s_t, a_t, s_{t+1}, r_{t+1})$, and in the case where action a_t executed in state s_t is the optimal action for Q_t — that is for $a_t = \pi_{Q_t}(s_t) = \text{argmax}_b\, Q_t(s_t, b)$ —, then the error term is:

$$r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)$$

which is exactly that of TD(0). This can be generalised to $\lambda > 0$, through coupling TD(λ) and Q-learning methods.

The SARSA (λ) algorithm [RUM 94] is a first illustration. As shown in Algorithm 2.2, it adapts Equation (2.17) to an action value function representation.

Algorithm 2.2: SARSA(λ)

```
/* α_t is a learning rate                                   */
Initialise(Q_0)
z_0 ← 0
s_0 ← ChooseState
a_0 ← ChooseAction
for t ← 0 until T_tot − 1 do
    (s'_t, r_{t+1}) ← Simulate(s_t, a_t)
    a'_t ← ChooseAction
    {update Q_t and z_t:}
    begin
        δ_t ← r_{t+1} + γQ_t(s'_t, a'_t) − Q_t(s_t, a_t)
        z_t(s_t, a_t) ← z_t(s_t, a_t) + 1
        for s ∈ S, a ∈ A do
            Q_{t+1}(s, a) ← Q_t(s, a) + α_t(s, a)z_t(s, a)δ_t
            z_{t+1}(s, a) ← γλz_t(s, a)
    end
    if s'_t non absorbing then
        s_{t+1} ← s'_t and a_{t+1} ← a'_t
    else
        s_{t+1} ← ChooseState
        a_{t+1} ← ChooseAction
return Q_{T_tot}
```

The $z_t(s, a)$ eligibility trace is extended to state-action pairs and the state space exploration is guided by the dynamics of the system, unless a terminal state is reached.

2.5.7. $Q(\lambda)$

Dealing with the case where the optimal action $\pi_{Q_t}(s'_t)$ was not chosen leads to $Q(\lambda)$ algorithms proposed by Watkins (see [SUT 98]) and Peng [PEN 94].

Watkins' approach to $Q(\lambda)$ is characterized by the fact that it only considers $\lambda > 0$ along pieces of trajectories where the current policy π_{Q_t} has been followed. Thus, with respect to SARSA(λ), the update rules of Q_t and z_t are both modified, as shown in Algorithm 2.3. The problem with this approach is that, when the exploration rate is high, the z_t traces are often reset to 0 and Q(λ) behaves closely to the original Q-learning.

Peng's approach to $Q(\lambda)$ solves this problem. It manages not to reset the trace z_t to 0 when an exploratory, non optimal action is chosen. There are very few experimental results about this approach (see however [NDI 99]) nor comparison between TD(λ), SARSA(λ) and Q(λ), thus drawing conclusions about these approaches is difficult. The only general conclusion that can be drawn is that using eligibility traces with $\lambda > 0$ reduces the number of necessary learning iterations to converge. The analysis in terms of computation time is more difficult, given the overhead resulting from the increased complexity of the algorithms.

Algorithm 2.3: Q(λ)

/* α_t is a learning rate */
Initialise(Q_0)
$z_0 \leftarrow 0$
$s_0 \leftarrow$ ChooseState
$a_0 \leftarrow$ ChooseAction
for $t \leftarrow 0$ **until** $T_{tot} - 1$ **do**
    $(s'_t, r_{t+1}) \leftarrow$ Simulate(s_t, a_t)
    $a'_t \leftarrow$ ChooseAction
    {update Q_t and z_t:}
    begin
        $\delta_t \leftarrow r_{t+1} + \gamma \max_b Q_t(s'_t, b) - Q_t(s_t, a_t)$
        $z_t(s_t, a_t) \leftarrow z_t(s_t, a_t) + 1$
        for $s \in S, a \in A$ **do**
            $Q_{t+1}(s,a) \leftarrow Q_t(s,a) + \alpha_t(s,a) z_t(s,a) \delta_t$
            $z_{t+1}(s,a) \leftarrow \begin{cases} 0 & \text{if } a'_t \neq \pi_{Q_t}(s'_t) \\ \gamma\lambda z_t(s,a) & \text{if } a'_t = \pi_{Q_t}(s'_t) \end{cases}$
    end
    if s'_t *non absorbing* **then**
        $s_{t+1} \leftarrow s'_t$ and $a_{t+1} \leftarrow a'_t$
    else
        $s_{t+1} \leftarrow$ ChooseState
        $a_{t+1} \leftarrow$ ChooseAction
return $Q_{T_{tot}}$

2.5.8. *The R-learning Algorithm*

All algorithms presented so far were based on the discounted criterion. The R-learning algorithm, proposed by Schwartz [SCH 93], is the adaptation of Q-learning to the average criterion and all previously explained principles are present.

The goal of this algorithm is to learn a policy π whose average reward ρ_π is as close as possible to the maximal average reward ρ^* of an optimal policy π^*. To ensure this property, R-learning maintains two correlated sequences ρ_t and R_t. The ρ_t sequence is only updated when the performed action was the greedy-action maximising R_t in the current state s_t. This ρ_t sequence is an estimate of the criterion to maximise. As Q_t in Q-learning, R_t represents the relative value function U of a policy:

DEFINITION 2.4.– *R value function*
We associate a new function R with a fixed policy π whose value function is U^π and whose average reward is ρ_π,:

$$\forall s \in S, a \in A \quad R^\pi(s,a) = (r(s,a) - \rho_\pi) + \sum_{s'} p(s'|s,a)U^\pi(s').$$

Here again, $U^\pi(x) = R^\pi(x, \pi(x))$ and the Bellman equation applied to ρ^* and R^* becomes:

$$\forall s \in S, a \in A \quad R^*(s,a) = (r(s,a) - \rho^*) + \sum_{s'} p(s'|s,a) \max_b R^*(s',b) \quad (2.18)$$

with a guarantee that the average reward of policy $\pi_{R^*}(s) = \operatorname{argmax}_a R^*(s,a)$ is the optimal reward ρ^*.

As Q-learning, R-learning is a stochastic version of value iteration for Equation (2.18).

Though there is no formal proof of the convergence of R-learning to the optimal policy, numerous experiments show that ρ_t efficiently approximates ρ^* under the same constraints as Q-learning.

Though R-learning is less famous and less used than Q-learning, it seems to perform more efficiently in practice [MAH 96b]. Few theoretical results are available on this point, but it has been shown that, in the finite horizon case, R-learning is very close to a parallel optimised version of Q-learning [GAR 98], which may explain its better empirical results.

Other average reward reinforcement learning algorithms have been proposed [MAH 96b]. Among them, the B algorithm [JAL 89] is an indirect method based on an adaptive estimation of the $p()$ and $r()$ functions. We can also mention Mahadevan's work

Algorithm 2.4: R-learning

```
/* α_t and β_t are learning rates */
Initialise(R_0,ρ_0)
for t ← 0 until T_tot − 1 do
    s_t ← ChooseState
    a_t ← ChooseAction
    (s'_t, r_t) ← Simulate(s_t, a_t)
    {update R_t and ρ_t:}
    begin
        R_{t+1} ← R_t
        δ_t ← r_t − ρ_t + max_b R_t(s'_t, b) − R_t(s_t, a_t)
        R_{t+1}(s_t, a_t) ← R_t(s_t, a_t) + α_t(s_t, a_t)δ_t
        ρ_{t+1} ← { ρ_t            if a_t ≠ π_{R_t}(s_t)
                  { ρ_t + β_t δ_t  if a_t = π_{R_t}(s_t)
    end
return R_{T_tot}, ρ_{T_tot}
```

[MAH 96a] who defined a model-based algorithm learning bias-optimal policies. Average reward reinforcement learning algorithms have been getting more popular recently (e.g. [BHA 07]).

We will now turn more generally to model-based approaches and we will present two algorithms that benefit from a proof of convergence in time polynomial in the size of the problem rather than exponential. These algorithms have also given rise to extension in the factored MDP case, as will be presented in Chapter 4.

2.6. Model-based Methods: Learning a Model

In Section 2.5.4, we highlighted the existence of a compromise between the learning speed and the memory usage. The solution based on eligibility traces is limited by the fact that learning results from information extracted from the immediate past of the agent. We will now present a different approach where the agent builds a model of its interactions with the environment and can then use dynamic programming algorithms based on this model, independently from its current state.

The main question with this kind of approach is the following: Shall we wait for the most exact possible model before performing Bellman backups? Shall we rather intertwin learning the model and applying dynamic programming on it? The latter approach seems more efficient, as illustrated by Adaptive Real-Time Dynamic Programming (ARTDP) [BAR 95], where model learning, dynamic programming and execution are concurrent processes. DYNA architectures such as *Dyna* [SUT 90a],

Queue-Dyna [PEN 93] and *Prioritized Sweeping* [MOO 93] are precursors of this approach.

2.6.1. *Dyna Architectures*

All model-based reinforcement learning algorithms are based on the same motivation. Rather than waiting for actual transitions of the agent in its environment, one way to accelerate the propagation of values consists in building a model of the transition and reward functions and to use this model to apply a propagation algorithm independently of the actual behaviour of the agent.

The DYNA architectures [SUT 90b], illustrated in Figure 2.4, were the first implementations of a system learning a model of transitions. Indeed, Sutton proposed this family of architectures to endow an agent with the capability to perform several updates at a given time step. To do so, several iterations of a dynamic programming algorithm are applied based on the model learned by the agent.

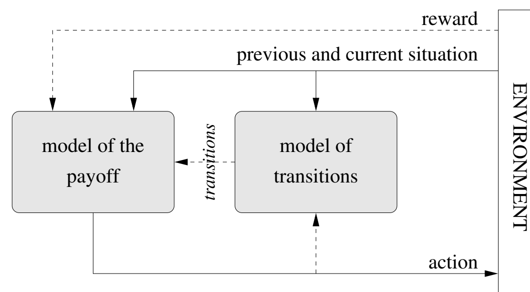

Figure 2.4. DYNA *architectures combine a model of the reward function and a model of the transition function. The model of transitions is used to accelerate learning. Building both models requires a memory of the previous situation, which is not explicit on the figure.*

The model of transitions is a list of $\langle s_t, a_t, s_{t+1}\rangle$ triples indicating that, after the agent performs action a_t in state s_t, it may reach state s_{t+1}. Once this model is learned, the value iteration or policy iteration algorithms can be applied on it. One can also perform "virtual transitions" using the model and a local update rule, any number of times per time step, so as to accelerate the propagation of values. This approach is particularly appealing when using physical systems like a robot, where performing an actual transition is much more expensive than a virtual one.

Furthermore, if the reward function suddenly changes, the value propagation mechanism can help reconfigure the whole value function quickly. Finally, building a model of transitions endows the agent with a planning capability: if one state is known as desirable or considered as a goal, the agent can look forward into its transition graph for

sequences of action that can result in reaching this state and execute the sequence that gives the highest probability of reaching it.

All *Dyna* architectures implement the principles described above. They differ from one another on the update rule or exploration strategy. The first implementation, DYNA-PI, also called *Dyna-AC* afterwards [SUT 98], is based on Policy Iteration.

Sutton [SUT 90b] showed that this system is less flexible than *Dyna-Q*, which is based on Q-learning. He finally proposed *Dyna-Q+*, a *Dyna-Q* augmented with active exploration capabilities.

Finally, the dynamic programming component of these systems can be improved by performing value updates more efficiently than a blind and systematic "sweep" of all transitions. Several improved sweeping mechanisms have been proposed, such as *Prioritized Sweeping* [MOO 93], *Focused Dyna* [PEN 92], *Experience Replay* [LIN 93] or *Trajectory Model Updates* [KUV 96].

2.6.2. *The E^3 Algorithm*

The first algorithm converging in polynomial time with respect to the size of the problem is called E^3, for *Explicit Explore and Exploit* [KEA 98]. Like DYNA architectures, E^3 builds a model of transitions and rewards. Nevertheless, instead of trying to build a complete model, it only memorises the subset of the visited transitions that are involved in the optimal policy.

To build this model, the algorithm visits all states homogeneously, i.e. the agent chooses in any state the less often performed action in that state. Then it updates a model of the probability distribution over subsequent states, which constitute a model of the transition function.

This way to deal with the exploration/exploitation trade-off is based on the "pigeon hole principle" that stipulates that, with such an homogeneous exploration, there will always be a time at which the model of the probability distribution for some state is close enough to the true transition function for that state. The authors define a notion of "known state" so that the number of visits necessary to consider a state as known remains polynomial in the size of the problem.

The model built by E^3 is an MDP containing all the "known" states with the observed transition probabilities and a unique absorbing state representing all the states that are not known yet. The algorithm then distinguishes two contexts:

– either there exists a policy that is close enough to the optimum based only on known states; in that case, dynamic programming can be applied to the known model;

– or this is not the case and the agent must keep exploring by taking actions leading to the absorbing state, so that more states get known.

This simple algorithm benefits from a proof of convergence in polynomial time to the optimal policy. The point is that, to decide in which of the above contexts it is, it must determine whether performing more exploration beyond a certain horizon would improve the current best policy, which is itself a hard problem.

2.6.3. *The $R_{\max}$ Algorithm*

The R_{max} algorithm [BRA 01] is an improvement over E^3. It relies on the same general idea of building a model of known states and looking for an optimal policy in the finite horizon case under an average reward criterion. But, furthermore, it simplifies the management of the exploration/exploitation trade-off thanks to an optimistic initialisation of the value function to the maximum expected immediate reward[3] and it extends the algorithm from the MDP framework to the null sum stochastic games framework, allowing to take the presence of an opponent into account.

The main difference with E^3 comes from the fact that, thanks to the optimistic initialisation of values, R_{max} does not need to decide whether it should explore or exploit. Indeed, the agent just goes towards attractive states, and a state is more attractive than the others either because it is not known enough or because it is along the optimal trajectory. With respect to E^3, R_{max} is more robust in the sense that, in the presence of an opponent, the agent cannot control accurately the transitions of the system — thus the homogeneous exploration strategy is difficult to ensure — whereas the optimistic exploration approach of R_{max} guarantees that the agent will explore as efficiently as possible whatever the behavior of the opponent.

Nonetheless, as for E^3, the proof of convergence of R_{max} calls upon two unrealistic assumptions. First, it is assumed that the algorithm knows the horizon T beyond which trying to improve the model through exploration will not result in significant improvements of the policy. Second, it is assumed that the optimal policy on horizon T over the known model can be computed at each time step, which is difficult in practice if T is large, as generally required in the context of a proof.

In practice, however, the authors show that the algorithm is efficient even with reasonable values of T, given that it performs better and better as T increases [BRA 03].

Other improvements have been proposed such as the heuristic sampling of transitions rather than a systematic exploration [PER 04, KEA 02].

---

3. Hence the name of the algorithm, R_{max}.

This polynomial time exploration topic has been very active in recent years, with work around Probably Approximately Correct (PAC) approaches, like MBIE [STR 05] and *Delayed Q-learning* [STR 06]. A recent synthesis of these approaches and further improvements are proposed in [SZI 08].

2.7. Conclusion

Reinforcement learning is nowadays a very active research domain, at the interface between machine learning, statistical learning, behaviour optimisation, robotics, cognitive sciences and even neurophysiology. Various techniques have been developed to study the acquisition of an optimal behaviour in a stochastic, dynamic and uncertain environment.

In this chapter, we have focused on the classical, basic methods that constitute the general background of the domain. This domain being very active, a lot of recent methods were not presented here.

Other classical approaches extending those presented here will be described in the next chapters of this book, such as on-line resolution methods (Chapter 6), dynamic programming with value function approximation (Chapter 3) or gradient methods to optimize parameterized policies (Chapter 5).

2.8. Bibliography

[BAR 83] BARTO A., SUTTON R., ANDERSON C. W., "Neuron-like Adaptive Elements That Can Solve Difficult Learning Control Problems", *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-13, num. 5, p. 834–846, 1983.

[BAR 95] BARTO A., BRADTKE S., SINGH S., "Learning to Act Using Real-time Dynamic Programming", *Artificial Intelligence*, vol. 72, p. 81–138, 1995.

[BER 95] BERTSEKAS D., *Dynamic Programming and Optimal Control*, Athena Scientific, Belmont, MA, 1995.

[BER 96] BERTSEKAS D., TSITSIKLIS J., *Neuro-Dynamic Programming*, Athena Scientific, Belmont, MA, 1996.

[BHA 07] BHATNAGAR S., SUTTON R. S., GHAVAMZADEH M., LEE M., "Incremental Natural Actor-Critic Algorithms", *Advances in Neural Information Processing Systems*, MIT Press, 2007.

[BRA 01] BRAFMAN R. I., TENNENHOLTZ M., "R-max: a General Polynomial Time Algorithm for Near-optimal Reinforcement Learning", *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI'01)*, p. 953–958, 2001.

[BRA 03] BRAFMAN R. I., TENNENHOLTZ M., "Learning to Coordinate Efficiently: A Model Based Approach", *Journal of Artificial Intelligence Research*, vol. 19, p. 11–23, 2003.

[CIC 95] CICHOSZ P., “Truncating Temporal Differences: On the Efficient Implementation of TD(λ) for Reinforcement Learning”, *Journal of Artificial Intelligence Research*, vol. 2, p. 287–318, 1995.

[DAY 94] DAYAN P., SEJNOWSKI T. J., “TD(λ) Converges with Probability 1”, *Machine Learning*, vol. 14, num. 3, p. 295–301, 1994.

[GAR 98] GARCIA F., NDIAYE S., “A Learning Rate Analysis of Reinforcement-Learning Algorithms in Finite-Horizon”, *Proceedings of the 15th International Conference on Machine Learning (ICML'98)*, San Mateo, CA, Morgan Kaufmann, p. 215–223, 1998.

[JAA 94] JAAKKOLA T., JORDAN M. I., SINGH S. P., “On the Convergence of Stochastic Iterative Dynamic Programming Algorithms”, *Neural Computation*, vol. 6, p. 1185–1201, 1994.

[JAL 89] JALALI A., FERGUSON M., “Computationally Efficient Adaptative Control Algorithms for Markov Chains”, *Proceedings of the IEEE Conference on Decision and Control (CDC'89)*, vol. 28, p. 1283–1288, 1989.

[KAE 93] KAELBLING L. P., *Learning in Embedded Systems*, MIT Press, Cambridge, MA, 1993.

[KEA 98] KEARNS M., SINGH S., “Near-Optimal Reinforcement Learning in Polynomial Time”, *Machine Learning*, vol. 49, 1998.

[KEA 02] KEARNS M. J., MANSOUR Y., NG A. Y., “A Sparse Sampling Algorithm for Near-Optimal Planning in Large Markov Decision Processes”, *Machine Learning*, vol. 49, num. 2-3, p. 193–208, 2002.

[KLO 72] KLOPF H. A., Brain Function and Adaptive Systems, A Heterostatic Theory, Report num. Technical Report AFCRL-72-0164, Air Force Cambridge Research Laboratories, 1972.

[KLO 75] KLOPF H. A., “A Comparison of Natural and Artificial Intelligence”, *SIGART newsletter*, vol. 53, p. 11–13, 1975.

[KUV 96] KUVAYEV L., SUTTON R. S., “Model-Based Reinforcement Learning with an Approximate, Learned Model”, *Proceedings of the 9th Yale Workshop on Adaptive and Learning Systems*, New Haven, CT, Yale University Press, p. 101–105, 1996.

[LIN 93] LIN L.-J., “Scaling Up Reinforcement Learning for Robot Control”, *Proceedings of the 10th International Conference on Machine Learning (ICML'93)*, Amherst, MA, Morgan Kaufmann, p. 182–189, 1993.

[MAH 96a] MAHADEVAN S., “An Average-Reward Reinforcement Learning Algorithm for Computing Bias-Optimal Policies”, *Proceedings of the National Conference on Artificial Intelligence (AAAI'96)*, vol. 13, 1996.

[MAH 96b] MAHADEVAN S., “Average Reward Reinforcement Learning: Foundations, Algorithms and Empirical Results”, *Machine Learning*, vol. 22, p. 159–196, 1996.

[MEU 96] MEULEAU N., Le dilemme entre exploration et exploitation dans l'apprentissage par renforcement, PhD thesis, Cemagref, université de Caen, 1996.

[MEU 99] MEULEAU N., BOURGINE P., "Exploration of Multi-State Environments: Local Measures and Back-Propagation of Uncertainty", *Machine Learning*, vol. 35, num. 2, p. 117–154, 1999.

[MIC 61] MICHIE D., "Trial and Error", *Science Survey*, vol. 2, p. 129–145, 1961.

[MIC 68] MICHIE D., CHAMBERS R., "BOXES: An Experiment in Adaptive Control", *Machine Intelligence*, vol. 2, p. 137–152, 1968.

[MOO 93] MOORE A., ATKESON C., "Prioritized Sweeping: Reinforcement Learning with Less Data and Less Real Time", *Machine Learning*, vol. 13, p. 103–130, 1993.

[NDI 99] NDIAYE S., Apprentissage par renforcement en horizon fini: application à la génération de règles pour la conduite de culture, PhD thesis, université Paul Sabatier, Toulouse, 1999.

[PEN 92] PENG J., WILLIAMS R., "Efficient Learning and Planning within the DYNA framework", MEYER J.-A., ROITBLAT H. L., WILSON S. W., Eds., *Proceedings of the 2nd International Conference on Simulation of Adaptive Behavior (SAB'92)*, Cambridge, MA, MIT Press, p. 281–290, 1992.

[PEN 93] PENG J., WILLIAMS R. J., "Efficient Learning and Planning within the Dyna Framework", *Adaptive Behavior*, vol. 1, num. 4, p. 437–454, 1993.

[PEN 94] PENG J., WILLIAMS R. J., "Incremental Multi-Step Q-learning", *Proceedings of the 11th International Conference on Machine Learning (ICML'94)*, vol. 11, p. 226–232, 1994.

[PEN 96] PENG J., WILLIAMS R. J., "Incremental Multi-Step Q-learning", *Machine Learning*, vol. 22, p. 283–290, Elsevier, 1996.

[PER 04] PERET L., GARCIA F., "On-line Search for Solving MDPs via Heuristic Sampling", *Proceedings of the European Conference on Artificial Intelligence (ECAI'04)*, 2004.

[RES 72] RESCORLA R. A., WAGNER A. R., "A Theory of Pavlovian Conditioning: Variations in the Effectiveness of Reinforcement and Nonreinforcement", BLACK A. H., PROKAZY W. F., Eds., *Classical Conditioning II*, p. 64–99, Appleton Century Croft, New York, NY, USA, 1972.

[RUM 94] RUMMERY G. A., NIRANJAN M., On-Line Q-learning using Connectionist Systems, Report , Cambridge University Engineering Department, Cambridge, Royaume-Uni, 1994.

[SAM 59] SAMUEL A., "Some Studies in Machine Learning using the Game of Checkers", *IBM Journal of Research Development*, vol. 3, num. 3, p. 210–229, 1959.

[SCH 93] SCHWARTZ A., "A Reinforcement Learning Method for Maximizing Undiscounted Rewards", *Proceedings of the 10th International Conference on Machine Learning (ICML'93)*, 1993.

[SIN 96] SINGH S. P., SUTTON R. S., "Reinforcement Learning with Replacing Eligibility Traces", *Machine Learning*, vol. 22, num. 1, p. 123–158, 1996.

[SIN 00] SINGH S. P., JAAKKOLA T., LITTMAN M. L., SZEPESVARI C., "Convergence Results for Single-Step On-Policy Reinforcement Learning Algorithms", *Machine Learning*, vol. 38, num. 3, p. 287–308, 2000.

[STR 05] STREHL A. L., LITTMAN M. L., "A theoretical analysis of Model-Based Interval Estimation", *Proceedings of the 22nd International Conference on Machine Learning (ICML'05)*, New York, NY, USA, ACM, p. 856–863, 2005.

[STR 06] STREHL A. L., LI L., WIEWIORA E., LANGFORD J., LITTMAN M. L., "PAC model-free reinforcement learning", *Proceedings of the 23rd International Conference on Machine Learning (ICML'06)*, New York, NY, USA, ACM, p. 881–888, 2006.

[SUT 81] SUTTON R., BARTO A., "Toward a Modern Theory of Adaptive Network: Expectation and Prediction", *Psychological Review*, vol. 88, num. 2, p. 135–170, 1981.

[SUT 88] SUTTON R., "Learning to Predict by the Method of Temporal Differences", *Machine Learning*, vol. 3, num. 1, p. 9–44, 1988.

[SUT 90a] SUTTON R. S., "Integrated Architectures for Learning, Planning and Reacting Based on Approximating Dynamic Programming", *Proceedings of the 7th International Conference on Machine Learning (ICML'90)*, San Mateo, CA, p. 216–224, 1990.

[SUT 90b] SUTTON R. S., "Planning by Incremental Dynamic Programming", *Proceedings of the 8th International Conference on Machine Learning (ICML'91)*, San Mateo, CA, Morgan Kaufmann, p. 353–357, 1990.

[SUT 98] SUTTON R. S., BARTO A. G., *Reinforcement Learning: An Introduction*, Bradford Book, MIT Press, Cambridge, MA, 1998.

[SZI 08] SZITA I., LŐRINCZ A., "The many faces of optimism: a unifying approach", *ICML '08: Proceedings of the 25th International Conference on Machine Learning*, New York, NY, USA, ACM, p. 1048–1055, 2008.

[THR 92] THRUN S., "The Role of Exploration in Learning Control", WHITE D., SOFGE D., Eds., *Handbook for Intelligent Control: Neural, Fuzzy and Adaptive Approaches*, Van Nostrand Reinhold, Florence, Kentucky, 1992.

[WAT 89] WATKINS C., Learning from Delayed Rewards, PhD thesis, Cambridge University, Cambridge, Royaume-Uni, 1989.

[WAT 92] WATKINS C., DAYAN P., "Q-learning", *Machine Learning*, vol. 8, num. 3, p. 279–292, Elsevier, 1992.