Chapter 1

Introduction

Nowadays every aspect of life is characterized by the need to make choices. Each choice is conditioned by a variable number of parameters. In many applicative domains (recommendation systems, medical analysis tools, real time game engines, speech recognizers...), this number could be very high. The possibility to manage in an efficient way such a number of parameters is guaranteed by a significant number of mathematical techniques increasingly performing and refined.

Every decisional problem can be translated into a *mathematical model* that represents the essence of the problem itself. A crucial step in formulating a model is the construction of the *objective function*. This requires developing a quantitative measure of performance relative to each of the decision maker's ultimate objectives that were identified while the problem was being defined. [HL01]

However we know that it is not always possible to define the objective function in advance. It is possible that objective function is unknown or, at least, partially unknown at the time the problem is dealt with. In these cases the optimization process of the objective function takes the name of *black-box optimization*.

Mathematically, we are considering the problem of finding a global maximizer (or minimizer) of an unknown objective function f :

$$x^* = \arg\max_{x \in \mathcal{X}} f(x) \tag{1.1}$$

where $\mathcal{X}$ is some design space of interest. In global optimization, $\mathcal{X}$ is often a compact subset of $\mathbb{R}^d$ [SSW+16].

Bayesian optimization is one of the best known black-box optimization techniques. It is a powerful tool for the joint optimization of design choices that is gaining great popularity in recent years. It is a sequential model-based approach to solving problem [AS08]. We prescribe a prior beliefs over the possible objective functions and then sequentially refine this model as data are observed via Bayesian posterior updating. The Bayesian Posterior represents our updated beliefs -given data- on the likely objective function we are optimizing [SSW+16].

In this thesis I propose an innovative approach to black-box optimization based on the Reinforcement Learning (RL) technique.

Reinforcement Learning (RL) is the problem faced by a learner that must behaviour through trial-and-error interactions with a dynamic environment. It can be considered the problem of mapping situations to actions in order to maximize a numerical reward signal [KLM96].

In the first part of this work I will compare the state of the art performances of the Bayesian model with those obtained through the implementation of a Reinforcement Learning (RL) agent in the same order.

In the last part of the thesis I will explain how RL and Bayesian optimization can be joined to emulate human brain learning process.

Chapter 2

Background

2.1 Markov Decision Process

Markov Decision Processes (MDPs) are a classical formalization of sequential decision making under uncertainty, where actions influence not just immediate responses, but also subsequent situations. Hence an MDP can be described as a controlled *Markov chain*[1], where the control is given at each step by the chosen action. In this chapter we will present the structure of an MDP and many techniques to solve its.

Markov Decision Process Markov decision processes are defined as controlled stochastic processes satisfying the *Markov property*[2] and assigning reward values to state transitions [Put94]. Formally, they are described by the 5-tuple (S, A, T, p, r) where:

- S is the state space in which the process' evolution takes place;
- A is the set of all possible actions which control the state dynamics;
- T is the set of time steps where decisions need to be made;
- $p()$ denotes the state transition probability function;
- $r()$ provides the reward function defined on state transitions.

[1] A sequence of random variables $X_0, X_1, ...$ with values in a countable set S is a *Markov chain* if at any time n, the future states (or values) $X_{n+1}, X_{n+2}, ...$ depend on the history $X_0, ..., X_n$ only through the present state X_n [Kon09] .

[2] A stochastic process has the *Markov property* if the probabilistic behaviour of the chain in the future depends only on its present value and discards its past behaviour.

States The set of environmental states S is defined as the finite set $\{s_1, ..., s_N\}$ where the size of the state space is N, i.e. $|S| = N$ [WvO12]. Each state $s \in S$ is a vector of attributes (*state variables*) that describes the current configuration of the system [NFK06]. More specifically, a state variable is the minimally dimensioned function of history that is necessary and sufficient to compute the "state of knowledge" [Pow07].

Actions The set of actions A is defined as the finite set $\{a_1, ..., a_K\}$ where the size of the action space is K, i.e. $|A| = K$. Actions can be used to control the system state. The set of actions that can be applied in some particular state $s \in S$, is denoted $A(s)$, where $A(s) \subseteq A$. In more structured representations the fact that some actions are not applicable in some states, is modelled by a precondition function : $S \times A \rightarrow true, false$, stating whether action $a \in A$ is applicable in state $s \in S$ [WvO12].

Time Steps The set of time steps T is defined as the finite set $\{t_1, ..., t_M\}$ where the size of the action space is M, i.e. $|T| = M$. Speaking about time we have to distinguish between *epochs* and *episodes*. An episode is made of a fixed number Z of epochs. An epoch is the smallest time unit in an MDP.

Transition Probability The transition probabilities $p()$ characterize the state dynamics of the system, i.e. indicate which states are likely to appear after the current state. For a given action a, $p(s'|s, a)$ represents the probability for the system to transit to state s' after undertaking action a in state s. This $p()$ function is usually represented in matrix form where we write P_a the $|S|$ x $|S|$ matrix containing elements $\forall s, s', P_{\text{a, s, s'}} = p(s'|s, a)$. Since each line of these matrices sums to one, the P_a are said to be stochastic matrices. The $p()$ probability distributions over the next state s' follow the fundamental property which gives their name to Markov decision processes. If we write $h_t = (s_0, a_0, ..., s_{\text{t-1}}, a_{\text{a-1}}, s_t)$ the history of states and actions until time step t, then the probability of reaching state $s_{\text{t+1}}$ consecutively to action a_t is only a function of a_t and s_t, and not of the entire history h_t [SB10]. We can resume this concept through the following equation :

$$\forall h_t, a_t, s_{\text{t+1}} \quad P(s_{\text{t+1}}|h_t, a_t) = P(s_{\text{t+1}}|s_t, a_t) = p(s_{\text{t+1}}|s_t, a_t) \tag{2.1}$$

Reward Function The reward function specifies rewards for being in a state, or doing some action in a state. The state reward function is defined as $R : S \rightarrow \mathbb{R}$, and it specifies the reward obtained in states. The

reward function is an important part of the MDP that specifies implicitly the *goal* of learning. Thus, the reward function is used to give direction in which way the system, i.e. the MDP, should be controlled [WvO12]. It is critical that the rewards we set up truly indicate what we want accomplished. If we reward the achievement of subgoals , then the agent might find a way to achieve them without achieving the real goal. Reward signal is our way of communicating the agent *what* we want to achieve, not *how* it achieved [SB18].

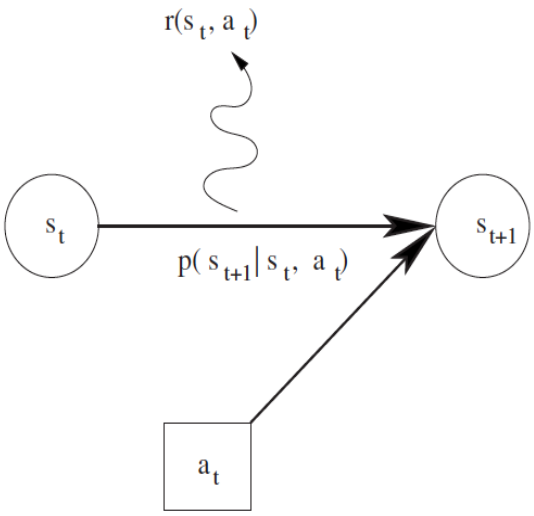

Figure 2.1: Markov decision process [SB10].

Markov decision processes allow us to model the state evolution dynamics of a stochastic system when this system is controlled by an agent choosing and applying the actions a_t at every time step t. The procedure of choosing such actions is called an action policy, or strategy, and is written as π [SB10] .

Policy Formally, given an MDP $\langle S, A, p(), r() \rangle$, a policy is a computable function that outputs for each state $s \in S$ an action $a \in A(s)$ [WvO12] A policy can decide deterministically upon the action to apply or can define a probability deistribution over the possible applicable actions. Then, a policy can be based on the whole history h_t (history-dependent policy) or can only consider the current state s_t. Thus, we can obtain four main families policies, as shown in table 2.1.

Policy π_t	Deterministic	Stochastic
Markov	$s_t \rightarrow a_t$	$a_t, s_t \rightarrow [0,1]$
History-dependent	$h_t \rightarrow a_t$	$h_t, s_t \rightarrow [0,1]$

Table 2.1: Different policy families for MDPs [SB10]

For a deterministic policy, $\pi_t(s_t)$ or $\pi_t(h_t)$ defines the chosen action a_t. For a stochastic policy, $\pi_t(a, s_t)$ or $\pi_t(a, h_t)$ represents the probability of selecting $a \in A$ for a_t [SB10]. The sets so defined are included in each other, from the most general case of stochastic, history-dependent policies, to the very specific case of deterministic, Markov policies, as shown in figure 2.2.

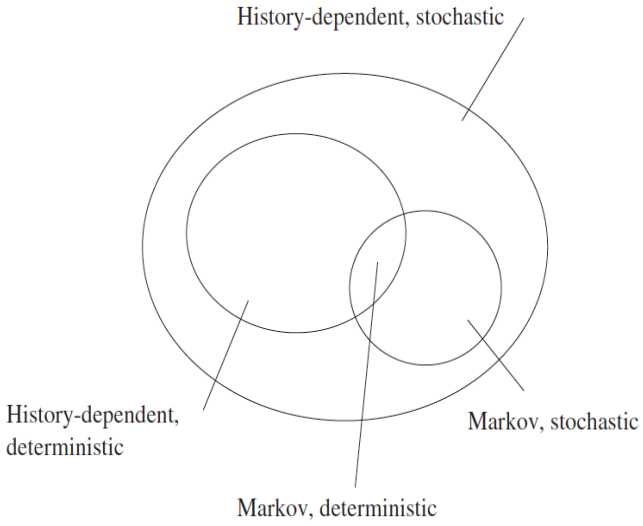

Figure 2.2: Relationship between the different sets of policies [SB10].

Application of a policy to an MDP is done in the following way. As shown in figure 2.3 each time an agent performs an action a_t in a state s_t, it receives a real-valued reward t_t that indicates the immediate value of this state-action transition. This produces a sequence of states s_i, actions a_i, and immediate rewards r_i as shown in the figure. The agent's task is to learn a control policy, $\{\pi : S \longrightarrow A\}$, that maximizes the expected sum of these rewards, with future rewards discounted exponentially by their delay [Mit97].

As already said the goal of learning in an MDP is to gather rewards. There are several ways of taking into account the future in how to behave

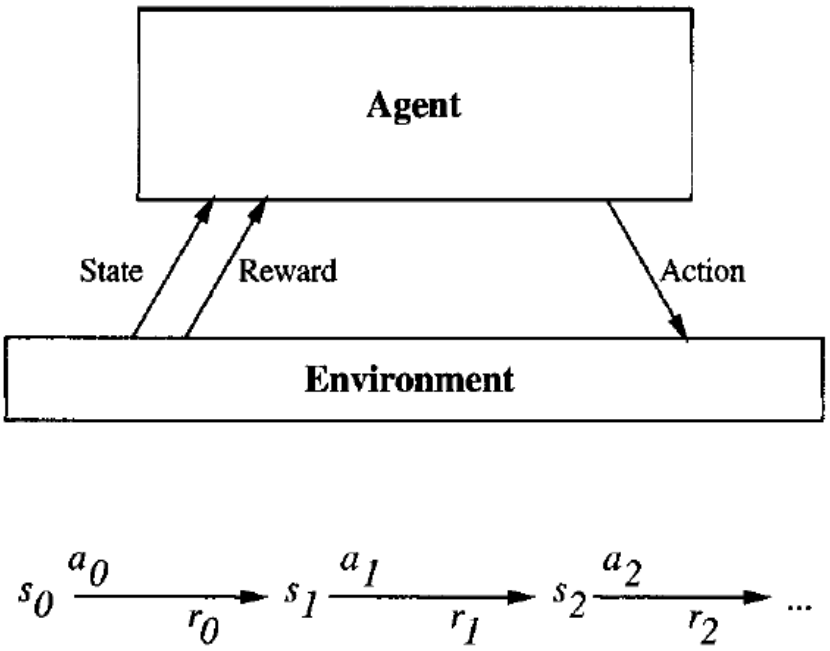

Figure 2.3: Policy application schema [SB18].

now. There are basically three models of optimality in the MDP, which are sufficient to cover most of the approaches in the literature :

$$E[\sum_{t=0}^{h} r_t] \qquad E[\sum_{t=0}^{\infty} \gamma^t r_t] \qquad \lim_{h\to\infty} E[1/h \sum_{t=0}^{h} r_t]$$

Figure 2.4: Optimality : **a)** finite horizon, **b)** discounted, infinite horizon, **c)** average reward

Discount Factor The *finite horizon* model simply takes a finite horizon of length h and states that the agent should optimize its expected reward over this horizon. In the *infinite-horizon model*, the long-run reward is taken into account, but the rewards that are received in the future are discounted according to how far away in time they will be received. A

discount factor γ, with $0 \leq \gamma \leq 1$ is used for this. Note that in this discounted case, rewards obtained later are discounted more than rewards obtained earlier. Additionally, the discount factor, ensures that, even with infinite horizon, the sum of the rewards obtained is finite. In episodic tasks, i.e. in tasks where the horizon is finite, the discount factor is not needed or can equivalently be set to 1. If $\gamma = 0$ the agent is said to be myopic, which means that it is only concerned about immediate rewards. The last optimality model is *average-reward* model, maximizing the long-run *average-reward*. Sometimes this is called the *gain optimal* policy and in the limit, it is equal to the infinite-horizon discounted model [WvO12]

Bellman Equation The concept of *value function* is the link between optimality criteria and policies. Most learning algorithms for MDPs compute optimal policies by learning value functions. The value of a state s under policy π, denoted $V^\pi(s)$ is the expected return when starting in s and following π thereafter. Using the infinite-horizon, discounted model :

$$V^\pi(s) = E_\pi\{\sum_{k=0}^{\infty} \gamma^k t_{\mathrm{t+k}} | s_t = s\} \tag{2.2}$$

A similar state-action value function : $Q : S \text{ x } A \rightarrow \mathbb{R}$ can be defined as the expected return starting from state s. taking action a and thereafter following policy π :

$$Q^\pi(s,a) = E_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{\mathrm{t+k}} | s_t = s, a_t = a\} \tag{2.3}$$

For any policy π and any state s the expression 2.3 can recursively be defined in terms of a so-called *Bellman Equation* :

$$\begin{aligned} V^\pi(s) &= E_\pi\{r_t + \gamma_{\mathrm{t+1}} + \gamma^2 r_{\mathrm{t+2}} + ... | s_t = t\} \\ &= E_\pi\{r_t + \gamma V^\pi(s_{\mathrm{t+1}}) | s_t = s\} \\ &\sum_{s'} T(s, \pi(s), s')(R(s,a,s') + \gamma V^\pi(s')) \end{aligned} \tag{2.4}$$

It denotes that the expected value of state is defined in terms of immediate reward and values of possible next state weighted by the transition probabilities, and additionally a discount factor. Note that multiple policies can have the same value function, but for a given policy π, V^π is unique. The goal for any MDP is to find a best policy, i.e. the policy that receives the

most reward. This means maximizing the value function of equation 2.3 for all states $s \in S$. An optimal policy, denoted π^*, is such that $V^{\pi^*}(s) \geq V^{\pi}(s)$ for all $s \in S$ and all policies π :

$$V^*(s) = \max_{a \in A} \sum_{s'} T(s, \pi(s), s')(R(s, a, s') + \gamma V^{\pi}(s')) \tag{2.5}$$

This expression is called the *Bellman optimality equation*. It states that the value of a state under an optimal policy must be equal to the expected return for the best action in a state [WvO12].

Value Iteration vs Policy Iteration Before studying how to solve MDPs let's clarify differences between two key concepts we will commonly find : *value iteration* and *policy iteration*. Policy iteration includes *policy evaluation* and *policy improvement*, and the two are repeated iteratively until policy converges. Value iteration includes finding *optimal value function* and extracting a policy; there is no repetition of those actions because onece the value function is optimal, than the policy should also be optimal, i.e. the policy should converges. Finding optimal value function can also be seen as a combination of policy improvements and truncated policy evaluation. The key step to policy improvements and policy extraction are identical except the former involves a stability check.

2.2 Solving MDP

Now that we have defined MDPs, policies, optimality criteria and value functions, it is time to consider the question of how to solve an MDP computing an optimal policy π^*. Several dimensions exists along which algorithms have been developed for this purpose. The most important distinction is that between *model-based* and *model-free* algorithms [WvO12].

	Model-based algorithms	**Model-free algorithms**
General name	DP	RL
Basic assumption	A model of the MDP is known beforehand, and can be used to compute value functions and policies using the Bellman equation.	Rely on interaction with the environment. Because a model of the MDP is not known, the agent has to explore the MDP to obtain information.
Planning	Yes	Yes
Learning	No	Yes

Table 2.2: Main differences between model-based and model-free algorithms.

2.2.1 Dynamic Programming

The term Dynamic Programming (DP) refers to a class of algorithms that is able to compute optimal policies in the presence of a perfect model of the environment described as a Markov Decision Process. These algorithms are known as *planning algorithms*. In planning, the idea is that we are given some description of a starting state or states; a goal state or states; and some set of possible actions that the agent can take; we want to find the sequence of actions that get us from the start state to the goal state. We search through a tree that is the sequences of actions that we can take, and we try to find a nice short plan. In so doing two of the possible planning algorithms are `BFS algorithm` and `DFS algorithm`.

The method of Dynamic Programming systematically records solutions for all sub-problems of increasing lengths. Using this programming paradigm the optimal policy is defined through the step-by-step definition of optimal sub-policies. According to Bellman optimality principle, all optimal sub-policies of an optimal policy are optimal sub-policies. DP algorithms are obtained by turning Bellman equations into update rules for improving approximations of the desired value functions.

Studying DP we can distinguish two different main methods. *Policy evaluation* refers to the typically iterative computation of the value functions for a given policy. *Policy improvement* refers to the computation of an improved policy given the value function for that policy. Either of these can be used to reliably compute optimal policies and value functions for finite MDPs given complete knowledge of the MDP.

Classical DP methods operate in sweeps through the state set, performing an *expected update* operation on each state. Each such operation updates the value of one state based on the values of all possible successor states and their probabilities of occurring. Expected updates are closely related to Bellman equations: they are little more than these equations turned into assignment statements. When the updates no longer result in any changes in value, convergence has occurred to values that satisfy the corresponding Bellman equation [SB18].

The assumption that a model is available will be hard to ensure for many applications, however DP algorithms are very relevant because they define fundamental computational mechanism which are also used when no model is available.

2.2.2 Reinforcement Learning

Dynamic Programming' s methods compute optimal policies for an MDP assuming that a perfect model is available. Reinforcement Learning (or *approximate dynamic programming*, or *neuro-dynamic programming*) is primarily concerned with how to obtain an optimal policy when such a model is not available. RL adds to MDPs a focus on approximation and incomplete information, and the need for sampling and exploration. In contrast with DP's algorithms, model-free methods do not rely on the availability of priori known transition and reward models. The lack of the model generates a need to *sample* the MDP to gather statistical knowledge about this unknown model. Many model-free RL techniques exist that probe the environment by doing actions, thereby estimating the same kind of state value and state-action value functions as model-based techniques [WvO12].

Roughly speaking Reinforcement Learning (RL) is the problem faced by a learner that must behaviour through trial-and-error interactions with a dynamic environment. It can be considered a problem of mapping situations to actions in order to maximize a numerical reward signal [KLM96].

In RL the learner must select an action to take in each time step: every choice done by the agent changes the environment in an unknown fashion and receives a reward which value is based on the consequences. The objective of the learner is to choose a sequence of actions based on observations of the current environment that maximizes cumulative reward or minimizes cumulative cost over all time steps [LM16]. The general class of algorithms that interact with the environment and update their estimates after each experience is called *online* RL.

Studying RL, one of the first problems we have to face with is the distinction between *direct* and *indirect* Reinforcement Learning. We will explain the difference between these two approaches in figure 2.5.

RL is different both from *supervised learning* and from *unsupervised learning*. It is not a sample of learning from a training set of labelled examples provided by a knowledgeable external supervisor (*supervised learning*) and it is not a sample of searching and finding structure hidden in a collection of unlabelled data (*unsupervised learning*). More specifically we can say that RL can be distinguished from other forms of learning based on the following characteristics :

- Reinforcement Learning deals with temporal sequences. In contrast with non-supervised learning problems where the order in which the

Algorithm 1: A general algorithm for online RL [WvO12]

```
1  foreach episode do
2      s ∈ S is initialized as the starting state;
3      t := 0;
4      repeat
5          choose an action a ∈ A(s);
6          perform action a ;
7          observe the new state s′ and received reward r update T̃, R̃, Q̃
             and/or Ṽ using the experience ⟨s, a, r, s′⟩;
8          s := s′;
9      until s′ is a goal state;
10 end
```

	REINFORCEMENT LEARNING	
	Model Based / Indirect	**Model Free / Direct**
Basic Assumption	First to learn the transition and reward model from interaction with the environment. After that, when the model is (approximately or sufficiently) correct, all the DP methods from the previous section apply.	Step right into estimating values for actions, without even estimating the model of the MDP.

Figure 2.5: RL's approaches classification.

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 a subsequent time steps [SB10].

- 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 [SB10].

According to what just said, one of the challenges that arise in RL, and not in other kind of learning, is the trade-off between *exploration* and *exploitation*. To obtain an higher reward, an RL agent must prefer actions that it has tried in the past and found to be effective in producing reward.

In order to discover such actions, it has to try actions that it has not selected before. The agent *exploits* what it has already experienced in order to obtain reward, but it has also to *explore* in order to eventually make better action selections in the future [SB18]. In other words *exploitation* consists of doing again actions which have proven fruitful in the past, whereas *exploration* consists of trying new actions, looking for a larger cumulated reward, but eventually leading to a worse performance. Dealing with the exploration/exploitation trade-off consists of determining how the agent should explore to get as fast as possible a policy that is optimal or close enough to the optimum. The most basic exploration strategy is the ϵ-greedy policy, i.e. the learner takes its current best action with probability $(1 - \epsilon)$ and a (randomly selected) other action with probability ϵ. [SB10].

Monte Carlo Methods Monte Carlo Methods (MC) represent a first instance of model-free RL's algorithms. They are one way solving the Reinforcement Learning problem based on averaging sample returns.

The Monte Carlo approach consists of 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.

More formally, let $(s_0, s_1, ..., s_N)$ be a trajectory consistent with the policy π and the unknown transition function $p()$, and let $(r_1, ..., r_N)$ be the rewards observed along this trajectory. In MC method, the N values $V(s_t), \quad t = 0, ..., N-1$ are updated according to :

$$V(s_t) \leftarrow V(s_t) + \alpha(s_t)(r_{\mathrm{t}+1} + r_{\mathrm{t}+2} + ... + r_N - V(s_t)) \tag{2.6}$$

with the learning rates $\alpha(s_t)$ converging to 0 along the iterations [SB10] . MC algorithms treat the long-term reward as a random variable and take as its estimate the sampled mean. Because the sampling is dependent on the current policy π, only returns for actions suggested by π are evaluated. Thus, *exploration* is of key importance here, just as in other model-free methods. One way of ensuring enough exploration is to use exploring starts, i.e. each state-action pair has a non-zero probability of being selected as the initial pair.

A distinction can be made between *every-visit* MC, which averages over all visits of a state $s \in S$ in all episodes, and *first-visit* MC, which averages

	On-policy methods	Off-policy methods
Basic assumption	They attempt to evaluate or improve the policy that is used to make decisions.	We do not need to follow any specific policy; our agent could even behave randomly and despite this it can still find the optimal policy.

Table 2.3: Difference between on-policy and off-policy methods.

over just the returns obtained from the first visit to a state $s \in S$ for all episodes. Both variants will converge to V^{π} for the current policy π over time [WvO12] .

Studying RL methods we have to distinguish between *on-policy methods* and *off-policy methods*. On-policy methods attempt to evaluate or improve the policy that is used to make decisions. Off-policy methods evaluate or improve a policy different from that used to generate the data. MC methods can be used for both on-policy and off-policy control, and the general pattern complies with the generalized policy iteration procedure.

Temporal Difference Learning An important problem faced by RL is the so called *temporal credit assignment problem*. In model-free contexts it is difficult to assess the utility of some action, if the real effects of this particular action can only be perceived much later. One possibility is to wait until the "end" (e.g. of an episode) and punish or reward specific actions along the path taken. However, this will take a lot of memory and often, with ongoing tasks, it is not known beforehand whether, or when, there will be an "end". Instead, one can use similar mechanisms as in value iteration to adjust the estimated value of a state based on the immediate reward and the estimated (discounted) value of the next state. This is generally called *temporal difference learning* which is a general mechanism underlying the model-free methods [WvO12].

We can formally represent this concept modifying equation 2.6 as follow:

$$V(s_t) \leftarrow V(s_t) + \alpha(s_t)(\delta_t + \delta_{\text{t+1}} + ... + \delta_{\text{N-1}}) \tag{2.7}$$

defining the temporal difference error δ_t by :

$$\delta_t = r_{\text{t+1}} + \gamma V(s_{\text{t+1}}) - V(s_t), \quad t = 0, ..., N-1. \tag{2.8}$$

The error δ_t must be interpreted in each state as a measure of the difference between the current estimation $V(s_t)$ and the correct estimation $r_{t+1} + V(s_{t+1})$.

Like Monte Carlo methods, TD ones performs estimation of action-value functions 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 TD methods, which are at the heart of most reinforcement learning algorithms, are characterized by this combination of estimation methods with local updates incremental properties [SB10].

Sarsa Algorithm : On-policy TD Control Knowing the exact value of all states is not always 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. To solve this problem, Watkins [WD92] introduced the action-value function Q, whose knowledge is similar to the knowledge of V when p is known. The action-value function of a fixed policy π whose value function is V^π is :

$$\forall s \in S, a \in A, \quad Q^\pi(s,a) = r(s,a) + \gamma \sum_{s'} p(s'\ s,a) V^\pi(s'). \tag{2.9}$$

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 :

$$\forall s \in S, a \in A \quad Q^*(s,a) = r(s,a) + \gamma \sum_{s'} p(s'|s,a) \max_b Q^*(s',b) \tag{2.10}$$

Than we have :

$$\forall s \in S, \quad V^*(s) = \max_a Q^*(s,a), \quad \pi^*(s) = \arg\max_a Q^*(s,a). \tag{2.11}$$

The SARSA algorithm works on state-action pairs rather than on states. Its update equation is :

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

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 : SARSA. This algorithm suffers from one conceptual drawback: performing the updates as stated above implies knowing 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 coubled to the current policy (the algorithm is called "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. Empirical studies often demonstrates the better performance of SARSA compared to Q-learning [SB10].

Algorithm 2: SARSA Algorithm : On-policy TD Control

```
Initialize Q(s, a) arbitrarily;
Repeat the next cycle until s is terminal;
foreach epsiode do
    Initalize s;
    Choose a from s using policy derived from Q (e.g. ε-greedy);
    foreach step of episode do
        Take action a, observer r, s';
        Choose a' from s' using policy derived from Q (e.g. ε-greedy);
        Q(s, a) ← Q(s, a) + α[r + γQ(s', a') − Q(s, a)] ;
        s ← s' ;
        a ← a' ;
    end
end
```

Q-learning Algorithm : Off-policy TD Control The Q-Learning Algorithm can be seen as a simplification of the algorithm, given that it is no more necessary to determine the action at the next step to calculate updates. Its update equation is :

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

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 the equation 2.12 is replaced by $\max_a Q(s_{t+1}, a)$ in equation 2.13. 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 $\alpha_t(s, a)$

Algorithm 3: Q-learning Algorithm [SB10]

α_t is the learning rate ;
Initialize (Q_0);
for $t = 0$; $t \leq T_{tot} - 1$; $t++$ **do**
    $s_t \leftarrow$ `ChooseState`;
    $a_t \leftarrow$ `ChooseAction`;
    $s_{t+1}, r_{t+1} \leftarrow$ `Simulate`(s_t, a_t);
    update Q_t ;
    begin
        $Q_{t+1} \leftarrow Q_t$;
        $\delta_t \leftarrow r_{t+1} + \gamma \max_b Q_t(s_{t+1}, b) - Q_t(s_t, a_t)$;
        $Q_{t+1}(s_t, a_t) \leftarrow Q_t(s_t, a_t) + \alpha_t(s_t, a_t)\delta_t$;
    end
end
return Q_{Tot}

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`. The `Initialize` function initializes the Q function with Q_0, which is often initialized with `null` values, whereas more adequate choices can highly improve the performance [SB10].

SARSA(λ) Previous algorithms only perform one update per time step in the state that the agent is visiting. 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 as a function of n before reaching the reward "trail". A naive way to solve the problem consists of using a memory of 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". A problem with this naive approach is that the required memory grows with length of trajectories, which is obviously not feasible in the infinite horizon context. In SARSA(λ) algorithm a more sophisticated

approach that addresses the infinite horizon context.

As we already saw Q-learning integrates the temporal difference error idea. With the update rule of Q-learning :

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha_t\{r_{t+1} + \gamma V_t(s_{t+1}) - Q_t(s_t, a_t)\} \tag{2.14}$$

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 optima action for Q_t, then the error term is $r_{t+1}+\gamma V_t s_{t+1}-V_t(s_t)$. For a generic parameter $\lambda \in [0, 1]$ we can generalize as follow:

where $z_t(s, a)$ and is the eligibility trace and *absorbing state* means terminal state.

Algorithm 4: SARSA (λ) Algorithm [SB10]

/* α is a learning rate */ ;
`Initialize` Q_0 ;
$z_0 \leftarrow 0$;
$s_0 \leftarrow$ `ChooseState` ;
$a_0 \leftarrow$ `ChooseAction` ;
$t \leftarrow 0$;
while $t \leq T_{tot} - 1$ **do**
    $(s'_t, r_{t+1}) \leftarrow$ `Simulate` (s_t, a_t) ;
    $a'_t \leftarrow$ `ChooseAction` ;
    update Q_t and z_t ;
    begin
        $\delta_t \leftarrow r_{t+1} + \gamma Q_t(s'_t, a'_t) - Q_t(s_t, a_t)$;
        $z_t(s_t, a_t) \leftarrow z_t(s_t, a_t) + 1$;
        for $s \in S, a \in A$ **do**
            $Q_{t+1}(s, a) \leftarrow Q_t(s, a) + \alpha_t(s, a) z_t(s, a) \delta_t$;
            $z_{t+1}(s, a) \leftarrow \gamma \lambda z_t(s, a)$
        end
        if s'_t *non absorbing* **then**
            $s_{t+1} \leftarrow s'_t$ and $a_{t+1} \leftarrow a'_t$;
        end
        else
            $s_{t+1} \leftarrow$ `ChooseState` ;
            $a_{t+1} \leftarrow$ `ChooseAction` ;
        end
    end
end
return Q_{Tot}

Bibliography

[AS08] Ryan P. Adams and Oliver Stegle. Gaussian process product models for nonparametric nonstationarity. In *ICML*, 2008.

[HL01] Frederick S. Hillier and Gerald J. Lieberman. *Introduction to Operations Research*. McGraw-Hill, New York, NY, USA, seventh edition, 2001.

[KLM96] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *CoRR*, cs.AI/9605103, 1996.

[Kon09] Takis Konstantopoulos. Markov chains and random walks. 2009.

[LM16] Ke Li and Jitendra Malik. Learning to optimize. *CoRR*, abs/1606.01885, 2016.

[Mit97] Tom M. Mitchell. *Machine Learning*. 1997.

[NFK06] Yuriy Nevmyvaka, Yi Feng, and Michael Kearns. Reinforcement learning for optimized trade execution. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 673–680, New York, NY, USA, 2006. ACM.

[Pow07] Warren B. Powell. *Approximate Dynamic Programming: Solving the Curses of Dimensionality (Wiley Series in Probability and Statistics)*. Wiley-Interscience, 2007.

[Put94] M. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley and Sons, 1994.

[SB10] Olivier Sigaud and Olivier Buffet. *Markov Decision Processes in Artificial Intelligence*. Wiley-IEEE Press, 2010.

[SB18] R. S. Sutton and A. G. Barto. *Reinforcement Learning : An Introduction*. 2018.

[SSW+16] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016.

[WD92] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[WvO12] M. Wiering and M. van Otterlo. *Reinforcement Learning: State-of-the-Art*. Adaptation, Learning, and Optimization. Springer Berlin Heidelberg, 2012.