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].

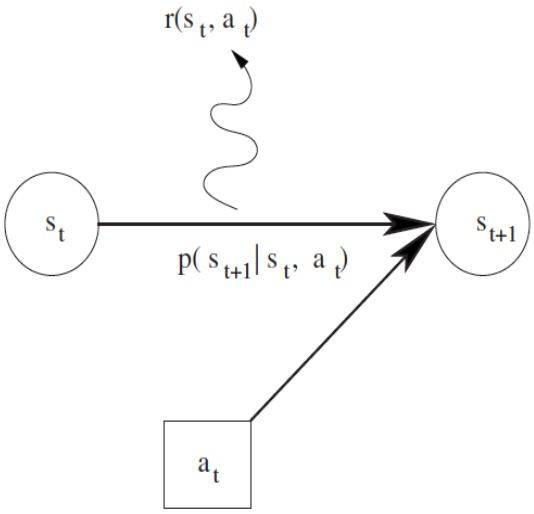


Figure 2.1: Markov decision process [SB10].

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.

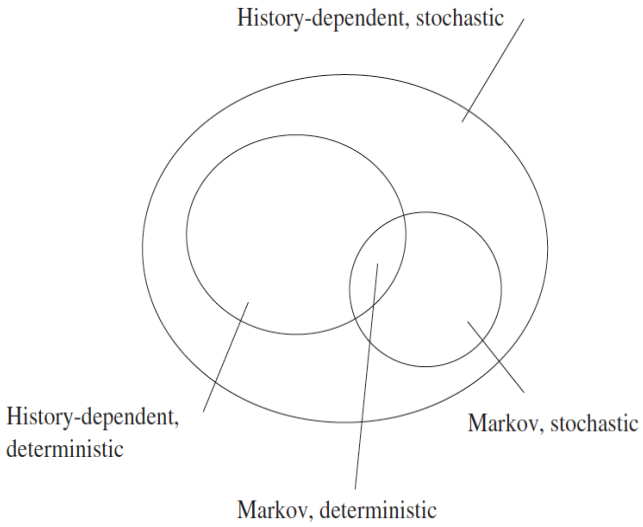


Figure 2.2: Relationship between the different sets of policies [SB10].

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

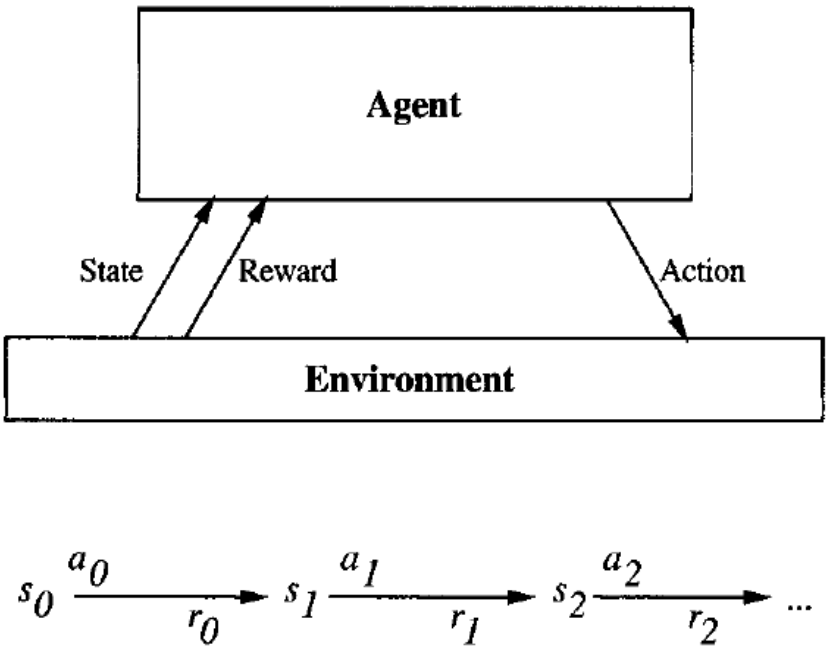


Figure 2.3: Policy application schema [SB18].

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 main differences between two key concepts we will recursively find : *value iteration* and *policy iteration*.

Both value iteration and policy iteration compute the same thing (all optimal values), i.e. they work with Bellman updates. In Value iteration we start with a random value function and then we find a new (improved) value function in an iterative process, until reaching the optimal value function. Value iteration computation of Bellman Equation is:

$$V_{\mathrm{k+1}}(s) \leftarrow \max_{a} \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V_k(s')] \tag{2.6}$$

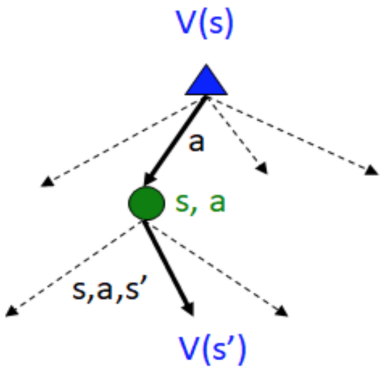


Figure 2.5: Value Iteration Schema

Value iteration has three disadvantages: it is slow ($O(S^2A)$ per iteration), the max at each state rerely changes, the policy often converges long

before the values.

Instead, policy iteration computations of Bellman Equation are:

- Evaluation : for a fixed current policy p, find values with policy evaluation:

$$V^{\pi_i}_{k+1}(s) \leftarrow \sum_{s'} T(s, \pi_i(s), s')[R(s, \pi_i(s), s') + \gamma V^{\pi_i}_{\mathrm{k}}(s')] \tag{2.7}$$

- Improvement : for fixed values, get a better policy using policy extraction :

$$\pi_{\mathrm{i+1}}(s) = \arg\max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^{\pi_i}(s')] \tag{2.8}$$

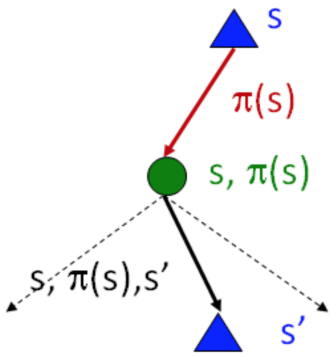


Figure 2.6: Policy Iteration Schema

Complexity of policy iteration is $O(S^2)$.

Summarizing what said until now we can write:

In value iteration :

- Every iteration updates both the values and implicitly the policy;
- We don't track the policy, but tacking the max over actions implicitly recomputes it.

In policy iteration :

- We do several passes that update utilities with fixed policy;
- After the policy is evaluated, a new policy is chosen;
- The new policy will be always better until the end.

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.

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

```
foreach episode do
    s ∈ S is initialized as the starting state;
    t := 0;
    repeat
        choose an action a ∈ A(s);
        perform action a ;
        observe the new state s' and received reward r update T̃, R̃, Q̃
          and/or Ṽ using the experience ⟨s, a, r, s'⟩;
        s := s';
    until s' is a goal state;
end
```

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 2.7.

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

	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 2.7: RL's approaches classification.

with 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 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 :

$$V(s_t) \leftarrow V(s_t) + \alpha(s_t)(r_{t+1} + r_{t+2} + ... + r_N - V(s_t)) \tag{2.9}$$

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 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.

	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.

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 2.9 as follow:

$$V(s_t) \leftarrow V(s_t) + \alpha(s_t)(\delta_t + \delta_{\text{t+1}} + ... + \delta_{\text{N-1}}) \tag{2.10}$$

defining the temporal difference error δ_t by :

$$\delta_t = r_{\text{t+1}} + \gamma V(s_{\text{t+1}}) - V(s_t), \quad t = 0, ..., N-1. \tag{2.11}$$

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 $r_{\text{t+1}} + V(s_{\text{t+1}})$.

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 :

$$\forall s \in S, a \in A, \quad Q^\pi(s,a) = r(s,a) + \gamma \sum_{s'} p(s'\ s,a) V^\pi(s'). \tag{2.12}$$

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 :

$$\forall s \in S, a \in A \quad Q^*(s,a) = r(s,a) + \gamma \sum_{s'} p(s'|s,a) \max_b Q^*(s',b) \tag{2.13}$$

Than we have :

$$\forall s \in S, \quad V^*(s) = \max_a Q^*(s,a), \quad \pi^*(s) = \arg\max_a Q^*(s,a). \tag{2.14}$$

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

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]. \tag{2.15}$$

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 :

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{2.16}$$

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 2.15 is replaced by $\max_a Q(s_{t+1}, a)$ in equation 2.16. 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`. 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

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}

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 :

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha_t\{r_{t+1} + \gamma V_t(s_{t+1}) - Q_t(s_t, a_t)\} \tag{2.17}$$

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:

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}

where $z_t(s, a)$ and is the eligibility trace and *absorbing state* means terminal state [SB10].

Chapter 3

Black-Box Optimization

One of the most exciting challenges in optimization practice is the possibility to optimize in absence of an algebraic model of the system to be optimized. This kind of optimization is known as *black-box optimization* and it is the subject matter of this dissertation.

A black-box function $f(x) : \mathbb{R}^n \to \mathbb{R}$ is a function for which the analytic form is not known. Nowadays there are lots of mathematical models to succeed in optimize this kind of functions. One of the best known of these methods is the *Bayesian Optimization Model* (BO).

In this chapter we will first analyse how BO works and than we will propose an innovative, Reinforcement Learning based approach to solve the same problem.

Gaussian Processes Before introducing BO we have to describe what *Gaussian Processes* (GPs) are. GPs are an alternative approach to regression problems. GP is a *non-parametric* approach (we don't have a priori knowledge of how many parameters will be useful for the regression) to find a distribution over the possible functions $f(x)$ that are consistent with observed data. A GP is a generalization of the Gaussian probability distribution. Whereas a probability distribution describes random variables which are scalars or vectors (for multivariate distributions), a *stochastic process* governs the properties of functions.

GP is a convenient and powerful prior distribution on functions, which we will take here to be of the form

$$f : \mathcal{X} \leftarrow \mathbb{R}. \tag{3.1}$$

The GP is defined by the property that any finite set of N points $\{x_n \in \mathcal{X}\}_{n=1}^{N}$ induces a multivariate Gaussian distribution on $\mathbb{R}^N$. The nth of these points is taken to be the function value $f(x_n)$ [SLA12]. The support and properties of the resulting distribution on functions are determined by a mean function

$$m : \mathcal{X} \leftarrow \mathbb{R} \tag{3.2}$$

and by a positive definite covariance function

$$k : \mathcal{X} \times \mathcal{X} \leftarrow \mathbb{R}. \tag{3.3}$$

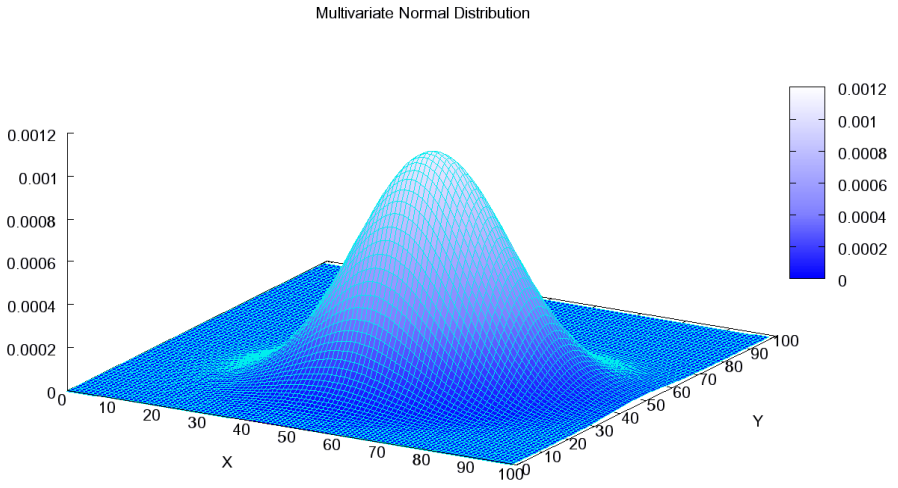


Figure 3.1: Multivariate Gaussian Distribution [MND].

Acquisition Functions for Bayesian Optimization Let's assume that the function $f(x)$ is drawn from a GP prior and that our observation are of the form $\{x_n \in \mathcal{X}\}_{n=1}^{N}$, where $y_n \sim \mathcal{N}(f(x_n), v)$ and v is the variance of noise introduced into the function observations. This prior and these data

induce posterior over functions; the acquisition function, which we denote by

$$a : \mathcal{X} \leftarrow \mathbb{R}^{+}, \tag{3.4}$$

determines what point in $\mathcal{X}$ should be evaluated next via a proxy optimization

$$x_{\text{next}} = \arg\max_{x} a(x), \tag{3.5}$$

where several different functions have been proposed. There are several popular choices of acquisition functions. Under the Gaussian process prior, these functions depend on the model solely through its predictive mean function $\mu(x; \{x_n, y_n\})$ and predictive variance function $\sigma^2(x; \{x_n, y_n\})$ [SLA12].

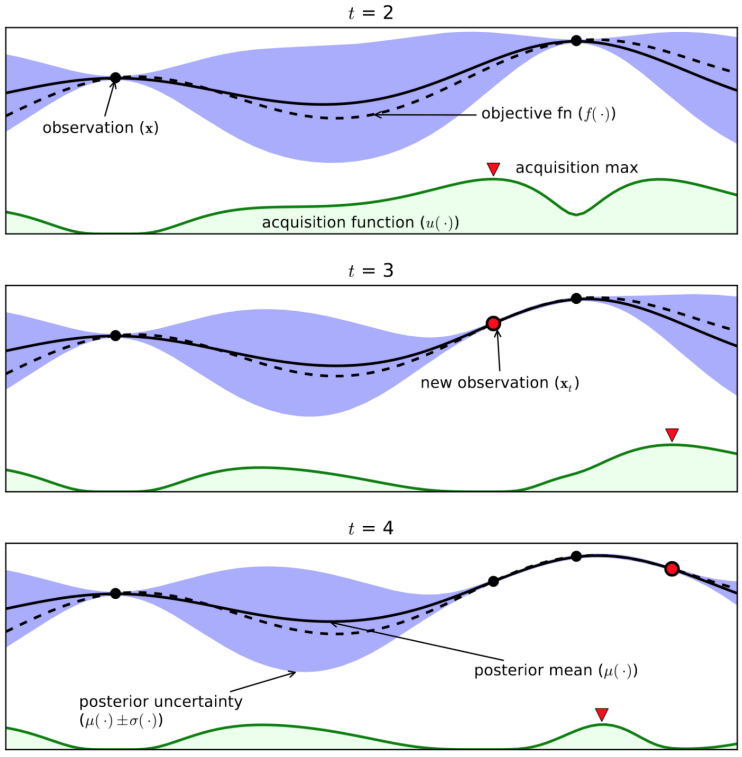


Figure 3.2: 2-*d* Bayesian Optimization Process Example [Bay]

An RL Approach To Black-Box Optimization As previously said, the aim of this thesis is to describe an innovative RL approach to the black-box function optimization problem. The process to reach this goal has required about three months of work. We made a lot of attempts. A full, detailed chronology of the development of the method presented in this chapter is described in **Appendix A**. Here we describe the final product of our efforts.

Let's consider the following scenario :

- **Agent** : The Reinforcement Learning agent has to maximize a black-box bivariate function. The function is continuously defined over a specific domain. The function is mapped on a space of 600×600 pixels. The agent has to complete its job having exactly 100 epochs for each one of the 150 episodes. In each epoch the agent has a specific position in space described through the two coordinates (x, y). Each time the agent makes an action the angle between (x, y) and (x', y') and the value of the function $f(x', y')$ are computed.

- **State** : The state is represented by two lists: the first one contains the last two computed *angles* and the second one contains the correspondent last two *actions*.

- **Actions** : In each epoch the agent can make one of four different actions : *move north*, *move south*, *move east*, *move west*. Each time the agent moves itself of 40 pixels in one of the previously described directions. The resultant effective movement over function is computed as shown in algorithm 5.

- **Reward** : In the context just described, a real terminal state doesn't exist. Cause we are working with black-box functions we cannot specify two coordinates (x, y) and the corresponding value function z as maximum, that is, as the terminal state. For this reason every action of the agent is rewarded. The simulation ends after 150 episodes are completed. In each explored state we define a Δ computed as follow:

$$\Delta = max f(x_n, y_n) - f(x, y) \tag{3.6}$$

where $f(x, y)$ is the value function computed in the *current state*. Reward at each epoch is equal to Δ.

Algorithm 5: From pixels to real values

```
1  /* knowing pixel_X and pixel_Y */;
2  /* knowing pixel_X_Range and pixel_Y_Range */ ;
3  /* knowing the function */;
4
5  domain = function.getDomain() ;
6  x_Range = domain.max_X - domain.min_X ;
7  y_Range = domain.max_Y - domain.min_Y ;
8
9  x_Real = domain.min_X + (pixel_X * x_Range) / pixel_X_Range ;
10 y_Real = domain.min_Y + (pixel_Y * y_Range) / pixel_Y_Range ;
11
12 return x_Real, y_Real
```

The nature of the movement the agent can make in each epoch can be of two different types : *linear movement* or *parametric movement*. A linear movement can be thought as a crow flies one. In this case the structure of the function is not really considered. On the other hand, a parametric movement can be thought as a real movement over the function curves. In this case the structure of the function influences the amount of the movement really done.

If the movement is linear we compute angles as follow :

Algorithm 6: Angle computation in linear movement case.

1 /* knowing (x, y) */ ;
2 /* knowing (x', y')*/ ;
3 /* `movementAmount` $= M$ */ ;
4 /* `currentMax` $= \max f(x_n, y_n)$ */ ;
5
6 $z = f(x, y)$;
7 $z' = f(x', y')$;
8
9 $\delta = ((x' - x), (y' - y))$;
10 $\alpha = \arctan(\frac{\delta}{\texttt{movementAmount}})$
11
12 Δ =z'−`currentMax`

We can explain this algorithm briefly recalling some trigonometry. Let's suppose to be in a simple $2d$-case like the one represented in figure 3.3.

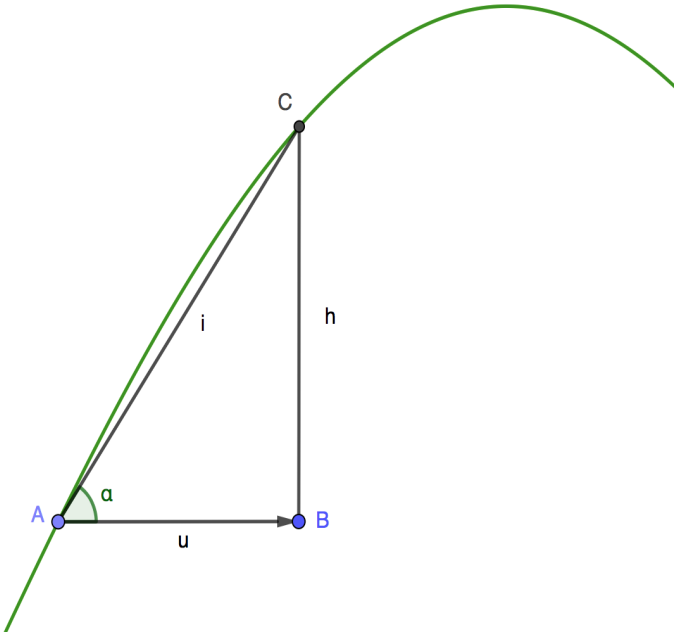


Figure 3.3: Linear movement computations.

Let's suppose that the starting point of our RL agent at epoch e is the point A with coordinates (x, y). In the $2d$-case the agent can make only one of two actions for each epoch : *move on* and *go back*. Making a linear movement of the type *move on* the arriving point at epoch *e'* is B with coordinates (x', y). Knowing that the objective function is $f(x) = \sin(2x)$, we can compute $f(x')$. At this time we should know point C with coordinates $(x', f(x'))$. u is the `movement amount` and $\overline{CB}$ equals to $f(x') - f(x)$ is the δ. From trigonometry

$$\tan\alpha = \frac{\delta}{\texttt{movementAmount}}, \tag{3.7}$$

so we can finally compute the angle α as follow :

$$\alpha = \arctan\frac{\delta}{movementAmount} \tag{3.8}$$

If the movement is parametric we have to be more specific about the amount of movement over the function. We do this using the following algorithm :

Algorithm 7: Angle computation in parametric movement case.

1 /* knowing (x, y) */ ;
2 /* knowing (x', y')*/ ;
3 /* `movementAmount` $= M$ */ ;
4 /* `currentMax` $= \max f(x_n, y_n)$ */ ;
5
6 $z = f(x, y)$;
7 $z' = f(x', y')$;
8
9 $\delta = ((x' - x), (y' - y))$;
10 $\alpha = \arctan(\frac{\delta}{\texttt{movementAmount}})$;
11
12 $\texttt{hypotenuse} = \frac{\texttt{movementAmount}}{\cos \alpha}$;
13
14 $\texttt{projectionOnHypotenuse} = \frac{\texttt{movementAmount} * \texttt{movementAmount}}{\texttt{hypotenuse}}$;
15
16 $\texttt{realMovementAmount} = \texttt{projectionOnHypotenuse} * \cos \alpha$;
17
18 Δ =z'−`currentMax`;

We can easily explain this algorithm looking at figure 3.4.

As previously done, let's suppose that the starting point of our RL agent at epoch e is the point A with coordinates (x, y). In the $2d$-case the agent can make only one of two actions for each epoch : *move on* and *go back*. Making a linear movement of the type *move on* the arriving point at epoch *e'* is B with coordinates (x', y). Knowing that the objective function is $f(x) = \sin(2x)$, we can now compute $f(x')$. We should now know point C with coordinates $(x', f(x'))$. u is the `movement amount` and CB equals to $f(x') - f(x)$ is δ. Using trigonometry we can compute α. Our objective is to know the real amount of movement done by the agent on the function. We want to know how much is $\overline{AE}$. From the first theorem of Euclid we know that *in a right-angled triangle, the square constructed on a cathetus is equivalent to the rectangle that has for dimensions the hypotenuse and the projection of*

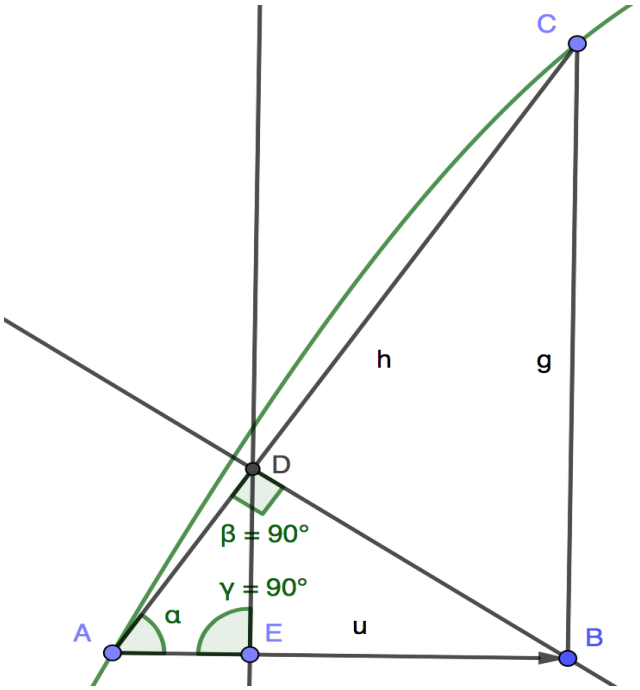


Figure 3.4: Parametric movement computations.

the cathetus on the hypotenuse. This means that in a right-angled triangle, the cathetus is proportional medium between the hypotenuse and its own projection on it. According to this we can write the following proportion :

$$h : u = u : AD. \tag{3.9}$$

So :

$$AD = \frac{u * u}{h}, \tag{3.10}$$

that is the same of :

$$\texttt{pojectionOnHypotenuse} = \frac{\texttt{movementAmount} * \texttt{movementAmount}}{\texttt{hypotenuse}} \tag{3.11}$$

of algorithm 7. Now we have all elements in order to compute the *real movement amount* represented by segment $\overline{AE}$. It is computed as :

$$\overline{AE} = \overline{DA} \cos \alpha \tag{3.12}$$

Both parametric and linear approach are effective. Therefore the question is why select one method instead of the other one. The answer to this question is simple. Adopting the parametric approach the optimization process is longer but more accurate. This depends on the fact that the real amount of movement depends on the angle so it is lesser each time.

On the other hand using the parametric approach, we could ideally train our agent on a set of specific functions and then it should have very good performances also on a generic, never previously explored, function using its knowledge about angles, actions and slope.

Adopting the linear approach the optimization process is slower but less accurate. This depends on the fact that the movement is as the crow flies. Using the linear approach we cannot abstract from the concept of function. This means that we have to train our agent for every specific function in order to optimize its. Any change on a specific function, however small, invalidates the previous training.

3.0.1 Implementation

In order to formalize and solve the problem previously described adopting an RL approach, we use the BURLAP (Brown-UMBC Reinforcement Learning and Planning) Java library developed and maintained by James MacGlashan of Brown University.

BURLAP uses a highly flexible system for defining states and and actions of nearly any kind of form, supporting discrete continuous, and relational domains. Planning and learning algorithms range from classic forward search planning to value function-based stochastic planning and learning algorithms [BUR].

In order to define customized MDPs, BURLAP offers a set of classes and interfaces (figure 3.5).

The main features offered by BURLAP are :

- **State** : implementing this interface we define the state variables of our MDP state space. An instance of this object will specify a single state from the state space [BUR] .

- **Action** : implementing this interface we define a possible action that the agent can select. If our MDP action set is discrete and unparametrized, we may consider using the provided concrete imple-

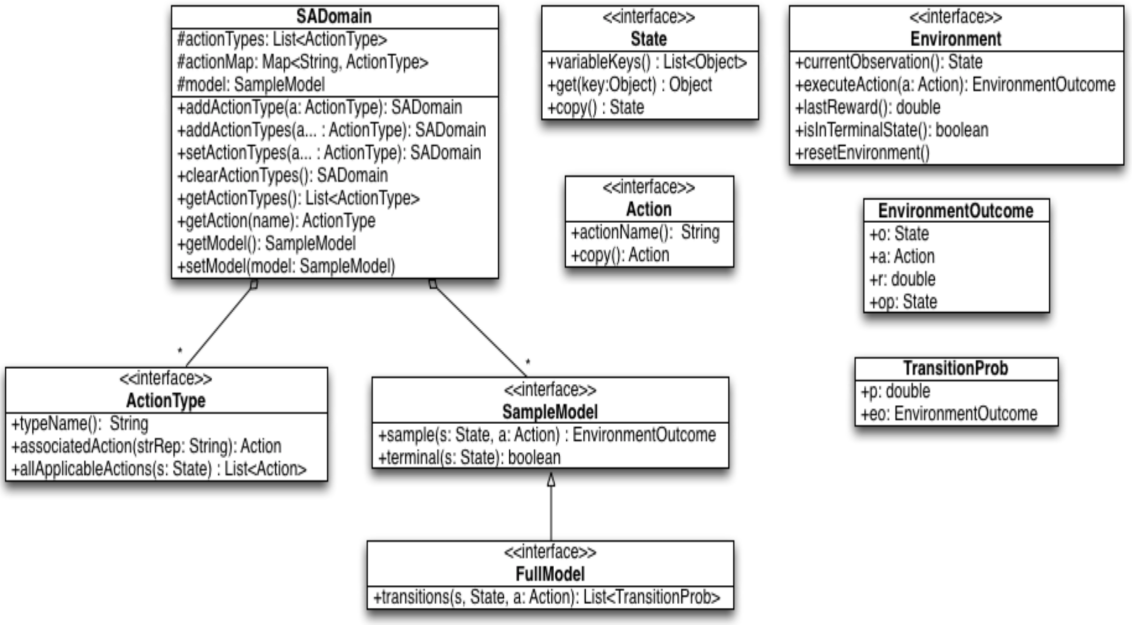


Figure 3.5: UML Digram of the Java interfaces/classes for an MDP definition.

mentation `SimpleAction`, which defines an action entirely by a single string name [BUR].

- **SampleModel** : implementing this interface we define the model of our MDP. This interface only requires us to implement methods that can sample a transition: spit back out a possible next state and reward given a prior state and action taken [BUR].

- **Environment** : An MDP defines the nature of an environment, but ultimately, an agent will want to interact with an actual environment, either through learning or to execute a policy it computed from planning for the MDP. An environment has a specific state of the world that the agent can only modify by using the MDP actions. Implement this interface to provide an environment with which BURLAP agents can interact. If we define the MDP ourself, then we'll probably don't want to implement `Environment` ourself and instead use the provided concrete `SimulatedEnvironment` class, which takes an `SADomain` with

a `SampleModel`, and simulates an environment for it [BUR].

An extended definition of classes and interfaces can be found at `http://burlap.cs.brown.edu/doc/index.html`.

in order to model the problem described above, starting from features offered by BURLAP, we extend interfaces and implement abstract classes as shown in figure 3.6.

Because of the detailed explanation of choices made to model the problem given in the first section of this chapter and in **Appendix A**, the mere analysis of architectural definition is left to the reader. The full code can be found at (...).

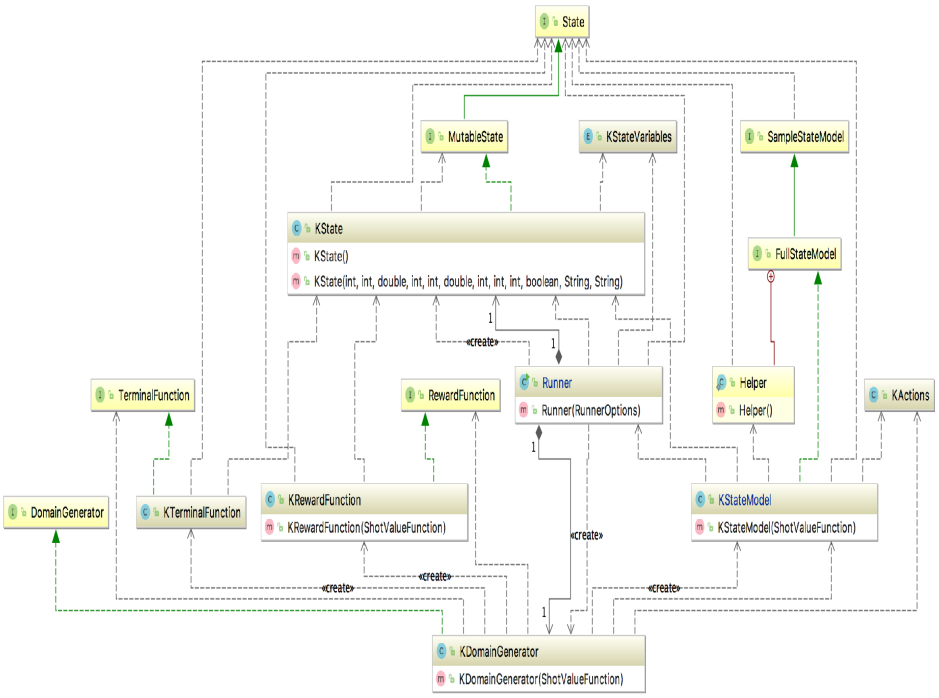


Figure 3.6: Class diagram of RL based optimization approach.

Chapter 4

Experimental Setting

The current chapter is divided into two sections. In the first one we will describe the benchmark of the experiment regarding the creation of the RL agent and its results. In the second one we will describe the same experiment applied to humans and its results.

4.1 Benchmark

Test Functions The experiment described in this work is about maximize black-box functions adopting an RL based approach. The first important choice is about selecting suitable *test functions*. In applied mathematics, test functions, also known as *artificial landscapes*, are useful to evaluate characteristics of optimization algorithms. We have chosen four test functions for this work:

- Himmelblau' s Function;
- Sphere Function;
- Beale Function;
- Styblinski-Tang' s Revised Function.

Himmelblau' s Function In mathematical optimization, Himmelblau' s function is a continuous, bivariate, multi-modal function introduced by David Mautner Himmelblau (1924–2011). The original function is defined by:

$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2 \tag{4.1}$$

It has four local minima :

- $f(3.0, 2.0) = 0.0$;
- $f(-2.805118, 3.131312) = 0.0$;
- $f(-3.779310, -3.283186) = 0.0$;
- $f(3.584428, -1848126) = 0.0$.

The function can be defined on any input domain but it is usually evaluated on $x \in [-5, 5]$ and $y \in [-5, 5]$.

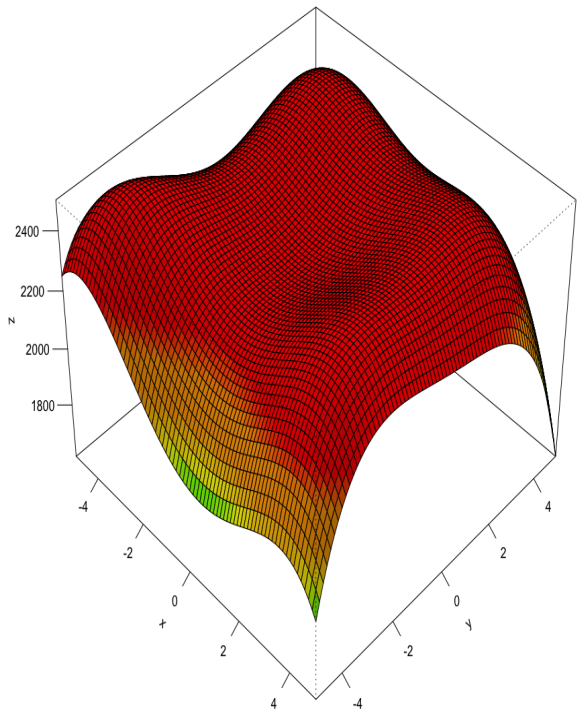


Figure 4.1: Customized Himmelblau' s Function.

Because of our aim to maximize we inverted the function as follow :

$$f(x,y) = -(x^2 + y - 11)^2 + (x + y^2 - 7)^2 \tag{4.2}$$

and we picked it up of 2500 units in order to have as less as possible negative values. So the final adopted function is :

$$f(x,y) = -(x^2 + y - 11)^2 + (x + y^2 - 7)^2 + 2500 \tag{4.3}$$

This function has its global maximum in $f(x,y) = 2500$.

In order to represent this customized version of Himmelblau' s Function using Java Graphical Environment, we mapped it in a space of 600×600 pixels and we properly rotated it. The resulting contour plot is the one represented in figure 4.2.

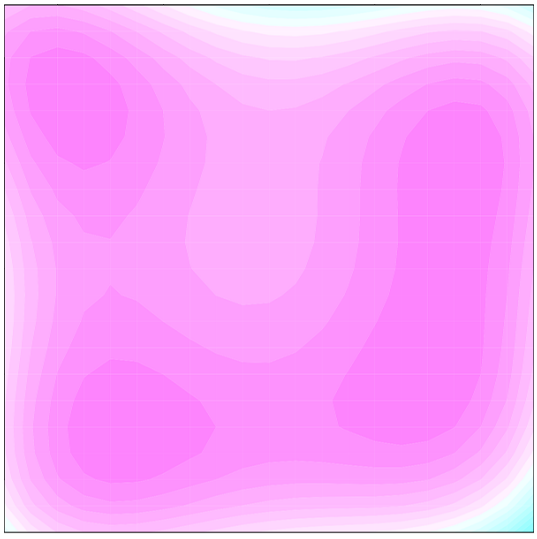


Figure 4.2: Contour plot of customized version of Himmelblau' s Function.

Paraboloid of Revolution In mathematical optimization, Paraboloid of Revolution is a continuous, bivariate, multi-modal function.

The original function is defined by:

$$f(x,y) = (x^2 + y^2) \tag{4.4}$$

It has a global minimum in $f(x,y) = 0$.

The function can be defined on any input domain but it is usually evaluated on $x \in [-10, 10]$ and $y \in [-10, 10]$.

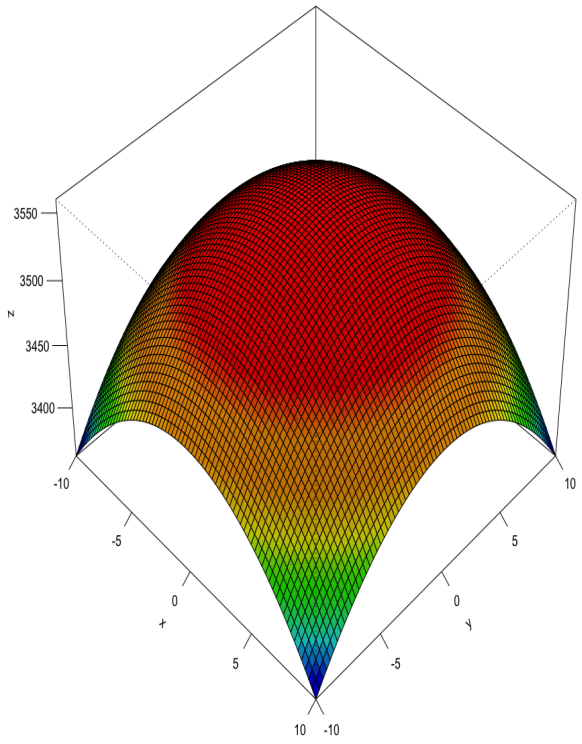


Figure 4.3: Customized Paraboloid of Revolution.

Because of our aim to maximize, we inverted the function as follow :

$$f(x, y) = -(x^2 + y^2) \tag{4.5}$$

and we picked it up of 3560 units in order to have as less as possible negative values. So the final adopted function is :

$$f(x, y) = -(x^2 + y^2) + 3560 \tag{4.6}$$

This function has its local maximum in $f(x, y) = 3560$.

In order to represent this customized version of Paraboloid of Revolution function using Java Graphical Environment, we mapped it in a space of 600×600 pixels and we properly rotated it. The resulting contour plot is the one represented in figure 4.4

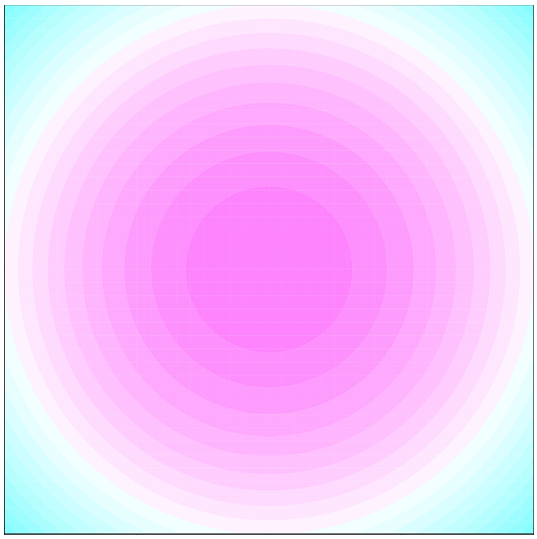


Figure 4.4: Contour plot of customized Parabolic Function.

Beale Function In mathematical optimization, Beale Function is a continuous, multi-modal function defined on a two-dimensional space. The function is defined by:

$$f(x,y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2 \quad (4.7)$$

The function can be defined on any input domain but it is usually evaluated on $x \in [-3, 3]$ and $y \in [-3, 3]$.

It has one global minimum at: $f(x, y) = 0$.

In this thesis our aim is to maximize. In order to do this we inverted the function as

$$f(x,y) = -((1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2) \quad (4.8)$$

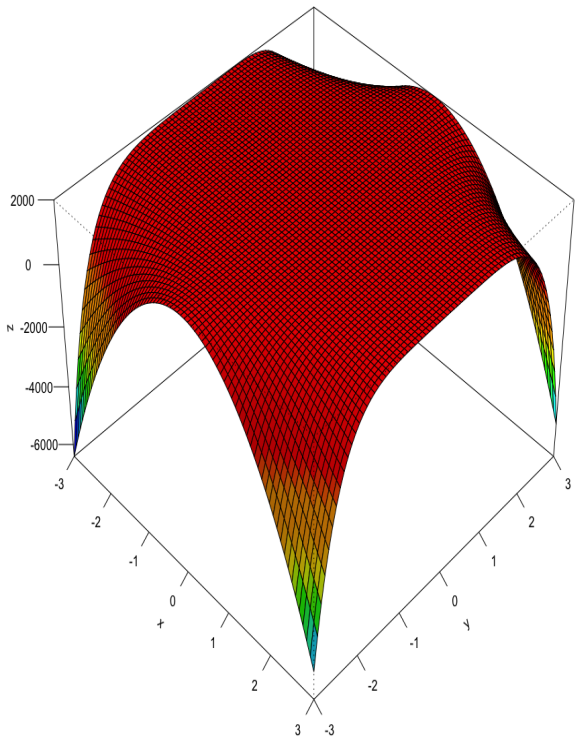


Figure 4.5: Customized Beale Function.

and we picked it up of 2000 units in order to have as less as possible negative values. So the final adopted function is :

$$f(x,y) = -((1.5-x+xy)^2+(2.25-x+xy^2)^2+(2.625-x+xy^3)^2)+2000 \quad (4.9)$$

This customized function has its global maximum in $f(x,y) = 1000$.

In order to represent this function using Java Graphical Environment we mapped it in a space of 600×600 pixels and we properly rotated it. The resulting contour plot is the one represented in figure 4.6

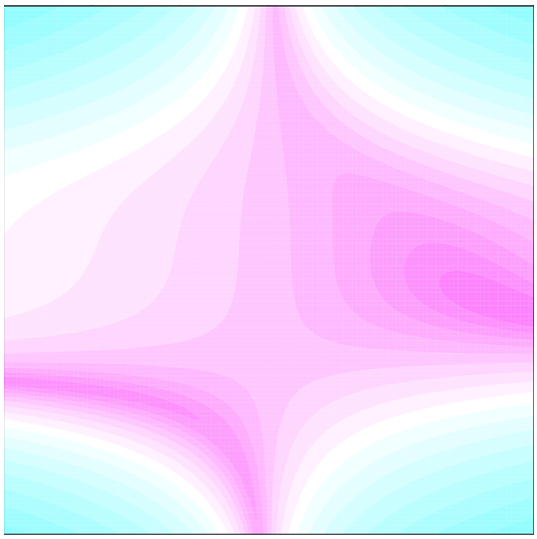


Figure 4.6: Contour plot of customized Beale Function.

Styblinski-Tang Revised Function In mathematical optimization, Styblinski-Tang Function is a continuous, multi-modal function defined on a multi-dimensional space. The original function is defined by :

$$f(x) = \frac{\sum_{i=1}^{n} x_i^4 - 16x_i^2 + 5x_i}{2} \tag{4.10}$$

In this thesis we consider the bivariate version of the original function multiplied by two in order to make it less flattened :

$$f(x, y) = (x^4 - 16x^2 + 5x) + (y^4 - 16y^2 + 5y). \tag{4.11}$$

The function can be defined on any input domain but it is usually evaluated on $x \in [-5, 5]$ and $y \in [-5, 5]$.

In this thesis our aim is to maximize. In order to do this we inverted the function as

$$f(x, y) = -((x^4 - 16 * x^2 + 5 * x) + (y^4 - 16 * y^2 + 5 * y)) \tag{4.12}$$

and we picked it up of 5000 units in order to has as less as possible negative values. So the final adopted function is:

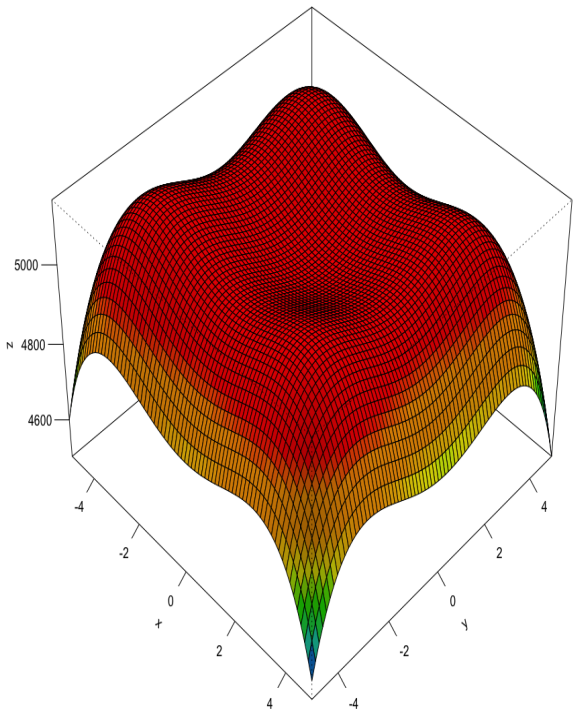


Figure 4.7: Customized Styblinski Function.

$$f(x,y) = -((x^4 - 16 * x^2 + 5 * x) + (y^4 - 16 * y^2 + 5 * y)) + 5000 \quad (4.13)$$

This customized function has its global maximum in $f(x,y) = 5156.6638$. In order to represent this function using Java Graphical Environment we mapped its in a space of 600×600 pixels and we properly rotated its. The resulting contour plot is the one represented in figure 4.8.

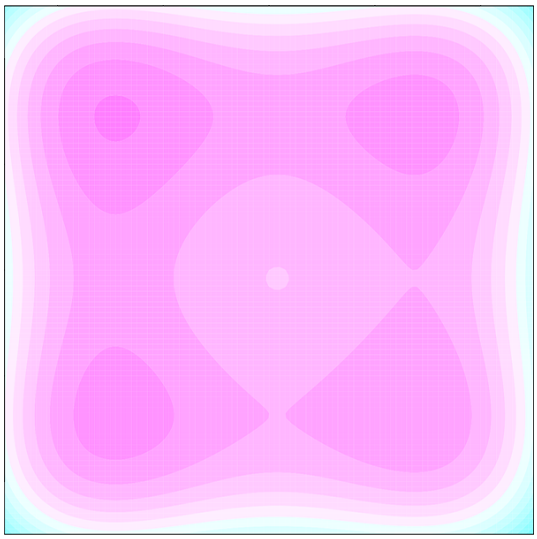


Figure 4.8: Contour plot of customized Styblinski Function.

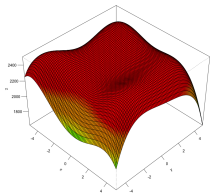
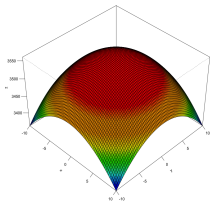
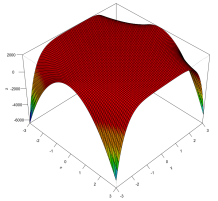
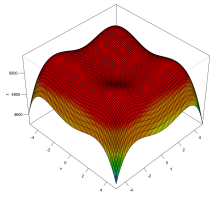
Algorithm Configuration The employed RL algorithm is the `SARSA(λ)` one. It has only one configuration (summarized in table 4.2) and four different declinations (summarized in table 4.3).

In the proposed configuration we use a low λ value. Typically we should use lower learning rates when we have a higher value of λ. However, in this case, we decide to use the highest possible learning rate. The logical consequence to this choice is to decrease the λ value.

An ϵ value equals to 0.1 means that 90% of selected actions depends on the history, whereas the remaining 10% represent a random choice having absolutely no relations with the previous history.

Speaking about the number of episodes and epochs, we have to say that our goal is to compare human optimization performances with `SARSA(λ)` ones in a context of uncertainty. Assuming that each try has an high cost, the number of possible tries must be as low as possible assuring, at the same time, a reasonable training space. In order to satisfy these requisites, the number of episodes is set to 150. Each episode is composed by 100 epochs. Note that in addition to the 150 training episodes we also have a greedy one. In the last episode the ϵ value and the *learning rate* parameter are set to 0. Instead, the number of epochs is left unchanged.

Table 4.1: Objective Functions' Summary Table

Name	Formula	Domain	Optimal f	Sketch
Himmelblau	$f(x,y) = -((x^2+y-11)^2+$ $+(x+y^2-7)^2) + 2500$	$x,y \in [-5;+5]$	2500	
Sphere	$f(x,y) = -(x^2+y^2) + 3560$	$x,y \in [-10;+10]$	3560	
Beale	$f(x,y) = -((1.5-x+xy)^2+)$ $+(2.25-x+xy^2)^2+$ $+(2.625-x+xy^3)^2) + 2000$	$x,y \in [-3;3]$	1000	
Styblinski-Tang	$f(x,y) = -((x^4-16*x^2+5*x)+$ $(y^4-16*y^2+5*y)) + 5000$	$x,y \in [-5;+5]$	5156.6638	

	λ	ϵ	**qInit**	**Learning Rate**	**Episodes**	**Epochs**
Configuration 1	0.1	0.1	0	1	150	100

Table 4.2: Configuration Table

Declination	**Type of search**
Declination 1	Linear, not random search
Declination 2	Linear, random search
Declination 3	Parametric, not random search
Declination 4	Parametric, random search

Table 4.3: Declination table

Each declination of `SARSA(λ)` corresponds to one combination of search techniques described in the previous chapter. In the implementation of random search we use the BURLAP `RandomFactory` class. `RandomFactory` allows us to logically group various random generators. The `id` is set to 0 and the `seed` is set to 1.

4.2 Human Case

The test conducted on humans affects a cross-section of thirty people. In the selection process of test subjects we considered the following three parameters with corresponding bounds :

- **Sex** $\in \{Male, Female\}$
- **Age** $\in [4, 52]$
- **Calculus Knowledge** $\in \{Null, Basic, Advanced\}$

Each of the test subjects was placed in front of a personal computer. The following rules have been proposed for individuals over the age of seven:

- *You will be offered four different levels.*
- *The first three levels will need to be repeated three times.*
- *The last level can be executed only once.*
- *Look carefully at the legend before starting the experiment.*

- *At each click you will receive a score with a colour associated with it.*
- *The goal is to make as many points as possible with 15 clicks available for each game.*
- *No further explanation will be provided.*

After that a chromatic scale indicating how positive the reward was for each click was proposed.

Figure 4.9: Chromatic Scale.

A white screen (with a specific hidden function) of 600×600 pixels with which to interact according to the rules previously listed was finally proposed . For those under the age of seven the selected approach was different. The rules have been explained to them verbally. At each interaction with the environment, in addition to the colour, they were verbally encouraged or discouraged.

Chapter 5

Results

This chapter is divided into three sections. In the first section we analyse results obtained from the experiment described in the first section of the previous chapter. For each function we compare results obtained applying the four different declinations of `SARSA(`λ`)` algorithm :

- Linear, not random search
- Linear, random search
- Parametric, not random search
- Parametric, random search

We employ three different plots for three different metrics. The first one measures the *Euclidean distance* [1] between point (x_i, y_i) at epoch e_i, $\forall i \in n$ where n is the number of epochs of greedy episode, and point (x^*, y^*) which is the maximum. The second metric measures the *proximity in percentage* [2] between the current value function and the optimal one in the greedy

[1] In mathematics, the Euclidean distance is the "ordinary" straight-line distance between two points in an Euclidean space. It is computed as follow:

$$Euclidean\_distance = \sqrt{(x_{\mathrm{i+n}} - x_i)^2 + (y_{\mathrm{i+n}} - y_i)^2} \tag{5.1}$$

with $n \geq 0$.

Note that this performance metric is evaluated on the true function values but this information is not available to the optimization methods.

[2] The *proximity in percentage* between the current value function and the optimal one is computed as follow:

episode. The third one represents the *gap metric*[3].

Himmelblau' s Function Plot (a) of figure 5.1 graphically represents performances of the four possible declinations of `SARSA(λ)` algorithm obtained considering the *Euclidean distance metric*.

Looking at the `linear, not random` declination's performance, we notice a starting high Euclidean difference between the agent's position and the global maximum. It decreases until epoch 5. Starting from this epoch, a recurrent pattern starts to be developed. This behaviour depends on the fact that once the agent has achieved a good position compared to the maximum, it has to continue to make movements until the terminal state is achieved. Each movement implies a reward. The agent selects a set of movements that maximizes the reward and systematically repeats them. The stability just described is also proved by the fact that the standard deviation of the current declination is the lowest of the set with a value of 25.3 (table 5.1).

`Linear, random` declination has a greater instability compared to the `linear, not random` one. In the first fifty epochs the *Euclidean distance* from the global maximum is constantly higher but then it starts to stabilize. Starting from the fiftieth epoch it develops a recurrent pattern and achieves best results. With an Euclidean distance mean of 71.6 pixels from the maximum, it has the best performance of the set (table 5.1).

The `parametric, not random` declination has worst performances of the declinations' set. It is unable to minimize the Euclidean distance from the global maximum and it is highly unstable. Looking at table 5.1 we note that the current declination as highest values in all statistic measures.

Finally, the `parametric, random` declination selects a starting point

$$Proximity\_in\_percentage = \frac{currentValueFunction \times 100}{optimalValueFunction} \tag{5.2}$$

Note that this performance metric is evaluated on the true function values but this information is not available to the optimization methods.

[3]This metric measures how effective each method is at finding the global maximum.

$$G_t = \frac{f(x^+, y^+) - f(x_1, y_1)}{f(x^* y^*) - f(x_1, y_1)} \tag{5.3}$$

Where x^+ is the incumbent or best function sample found up to epoch e. The gap G_t will therefore be a number between 0 (indicating no improvement over the initial sample) and 1 (if the incumbent is the maximum). Note that this performance metric is evaluated on the true function values but this information is not available to the optimization methods. [HBdF11]

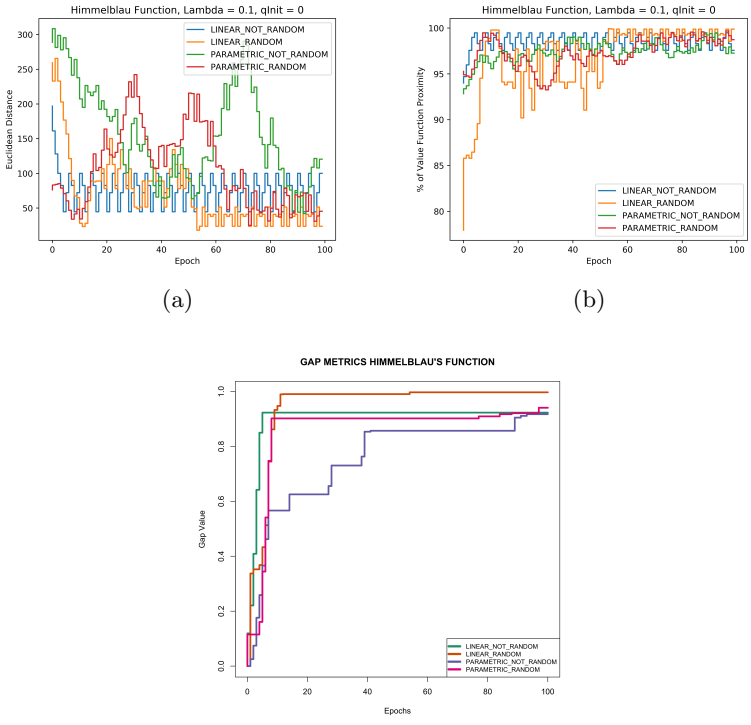


Figure 5.1: Himmelblau' s Function.

near to the global maximum but it is unable to maintain a low Euclidean distance from it until epoch seventy. Starting from this epoch it develops a recurrent pattern similar to one previously described. Despite of this we can say that its performances are worse compared to the first two ones.

Slower convergence of linear declinations compared to parametric ones depends on the fact that in the first case, as already explained in chapter 3, greater movements are done at each epoch and a lower training time is required to achieve the maximum. The amount of parametric movements are strictly conditioned by the function's shape. If this represents an obstacle to a fast convergence, however it is an incentive to an higher precision.

Plot (b) of figure 5.1 graphically represents performances of the four possible declinations of `SARSA(λ)` algorithm obtained considering the *Percentage of Value Function' s Proximity metric.*

Himmelblau' s Function	Mean-ED	Standard deviation - ED
Linear not Random	77.7	25.3
Linear Random	71.6	52.2
Parametric not Random	158.6	71.3
Parametric Random	105.5	56.0

Table 5.1: Euclidean Metric's Performances

All declinations, except for the `linear, random` one, start with an high level of value function's proximity to the maximum. It depends on the function's shape.

Once it has achieved a good enough proximal point to the maximum, the `linear, not random` declination maintains always the same vale function's proximity level developing a recurrent pattern. The average level of proximity achieved by this declination is the best of the set (table 5.2). The highly stability of the current declination is proved by the lowest standard deviation of the set (table 5.2).

The `linear, random` declination starts from a point not very close to the maximum in terms of value function's value. Its great instability reveals an inadequate training space. It starts to stabilize starting from the fiftieth epoch. Its starting instability is proved by the highest standard deviation of the set (table 5.2).

The `parametric, not random` declination reveals a general stability. The growth of proximity is slower then the corresponding linear declination because of the reduced amount of movement depending on the function's shape.

Finally, the `parametric, random` declination, as the corresponding linear declination, shows an initial instability. In general performances are more stable than the linear corresponding ones because of the lower amount of movement done at each epoch. Starting from the sixty-fifth epoch the proximity slowly grows and stabilizes.

Plot (c) of figure 5.1 graphically represents performances of the four possible declinations of `SARSA(`λ`)` algorithm obtained considering the *gap metric*. According to this plot the most effective declination of `SARSA(`λ`)` algorithm in finding the maximum is the `linear, random` one. Looking at table 5.9 we note that `linear, random` declination achieves point $(3.67, -1.57)$ with a value function of 2498.457 out of 2500.0.

Himmelblau' s Function	Mean- P	Standard deviation - P
Linear not Random	98.5	0.96
Linear Random	96.8	4.0
Parametric not Random	97.4	1.2
Parametric Random	97.3	1.6

Table 5.2: Value Function' s Proximity.

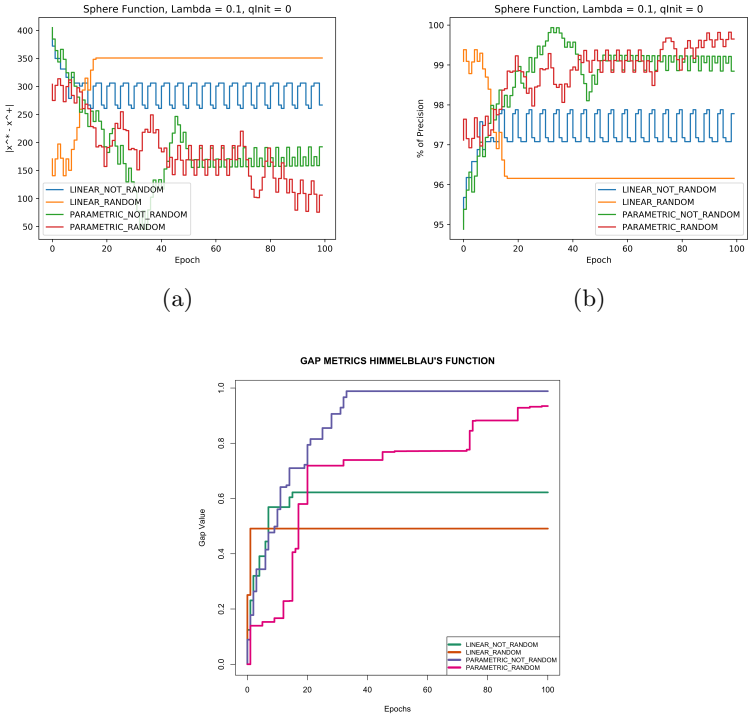


Figure 5.2: Sphere Function.

Sphere Function Before starting to analyse performances of different declinations of `SARSA(`λ`)` algorithm in maximizing the Sphere function, it is important to underline that the starting point for the current function in non-random declinations is $(0, 0)$. The reason for this choice is that $f(300, 300)$ is itself the maximum of the function.

Plot (a) of figure 5.2 graphically represents performances of the four possible declinations of `SARSA(λ)` algorithm obtained considering the *Euclidean distance metric*. Looking at the `linear, not random` declination we notice a constant, high Euclidean difference between the agent's position and the global maximum. This distance never decreases and still from the first epochs, a recurrent pattern starts to be developed. The inefficiency of the algorithm's declination depends on an inadequate training space. The relationship between different explored states and total possible states does not permit the agent to develop an efficient policy. Bad performance's results of the current declination are also certified by an high metric's average (table 5.3). On the other hand the great stability of this declination is underlined by the lowest standard deviation of the set (table 5.3).

`Linear, random` declination starts from a point relatively close to the maximum, but, one more time, the inadequate training space prevents its to develop a rally optimal policy. This declination is the less efficient one with an average Euclidean distance from the maximum of 328.9 pixels.

Better performances can be registered looking at parametric declinations of the algorithm. The `parametric, not random` declination starts with a physiological high Euclidean difference from the maximum. It rapidly decrease until the thirty-fifth epoch and then it has another soft increase. Starting from the fifty-fifth epoch a recurrent pattern starts to be developed. Despite of the fact that this declination is the most unstable one with a standard deviation of 67.9 (table 5.3), it has one of the lowest average Euclidean distance from the maximum.

The `parametric, random` declination of `SARSA(λ)` algorithm has the best performance of the set. After a starting, considerable instability it reaches an Euclidean difference of around fifty pixels. The good performance is also proved by the fact that the average Euclidean distance from the maximum is 185.3 pixels.

Sphere Function	**Mean-ED**	**Standard deviation - ED**
Linear not Random	293.2	25.6
Linear Random	328.9	56.3
Parametric not Random	190.7	67.9
Parametric Random	185.3	58.0

Table 5.3: Euclidean Metric's Performances

Plot (b) of figure 5.2 graphically represents performances of the four possible declinations of `SARSA(λ)` algorithm obtained considering the *Percentage of Value Function' s Proximity metric*.

The `linear, not random` declination starts from a value function's proximity of about 97.5% from the maximum. It is completely unable to improve this percentage. Already starting from the first epochs it develops a recurrent pattern. This behaviour allows its to have the lowest standard deviation of the declinations' set (table 5.4). Unfortunately the value function's proximity mean is the second lowest one.

The `linear, random` declination starts from a very high value function's proximity level. Because of an inadequate training space it decreases until the twentieth epoch. Starting from this epoch it stars to stabilize. Effects of this behaviour on the average value function's proximity to the maximum and on average standard deviation are relevant.

The `parametric, not random` declination is the most unstable one. It starts with a low value function's proximity level and slowly grows achieving peaks of about 100%. Starting from the fiftieth epoch it stabilizes developing a recurrent pattern. The initial high instability has effects on average value function's proximity variance making its the highest one (table 5.4).

The `parametric, random` declination of `SARSA(λ)` algorithm is the most efficient one. It starts with a value function's proximity level of about 97% and slowly grows first stabilizing around 99% and than growing more achieving peaks of 99.8%. Looking at table 5.4 we can notice that the average value function's proximity is the highest one with a level of 98.8%.

In this case we observe that parametric declinations do better than linear ones. In particular the random starting position allows the agent to develop a better policy. The main reason of the failure of non random declinations is the reduced training space compared to the distance between the starting point and the maximum.

Sphere Function	**Mean- P**	**Standard deviation - P**
Linear not Random	97.3	0.49
Linear Random	95.5	0.92
Parametric not Random	98.7	0.98
Parametric Random	98.8	0.72

Table 5.4: Value Function' s Proximity.

Plot (c) of figure 5.1 graphically represents performances of the four

possible declinations of `SARSA(λ)` algorithm obtained using the *gap metric*. According to this plot the most effective declination of `SARSA(λ)` algorithm in finding the maximum is the `parametric, not random` one. Looking at table 5.9 we note that `parametric, not random` declination achieves point $(-1.033, -1.099)$ with a value function of 3557.722 out of 3560.0.

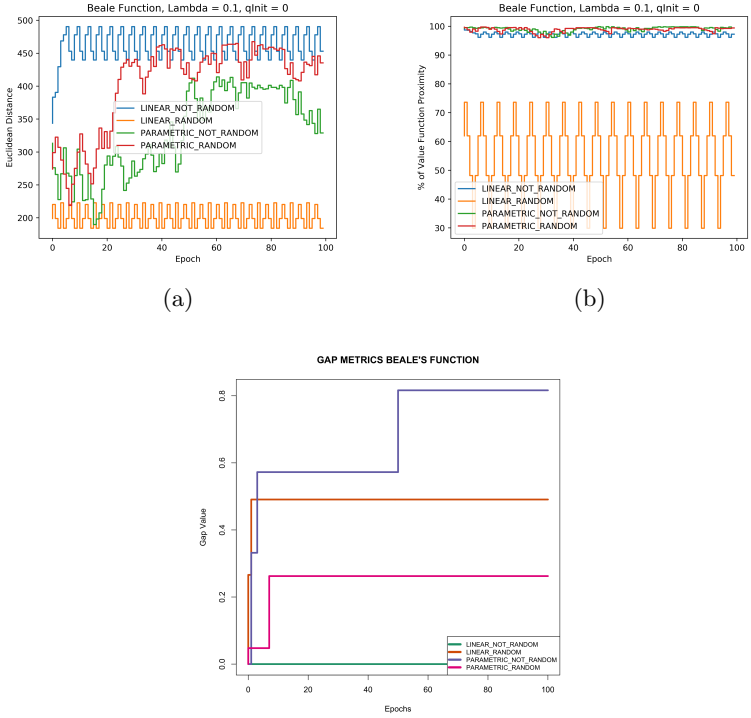


Figure 5.3: Beale Function.

Beale Function Giving an overview to figure 5.3 we can see that Beale Function represents the case in which `SARSA(λ)` algorithm' s declinations do worst.

Plot (a) of figure 5.3 graphically represents performances of the four possible declinations of `SARSA(λ)` algorithm obtained considering the *Euclidean distance metric*. Looking at the `linear, not random` declination we notice a growing performance' s worsening until tenth epoch. Starting from epoch ten algorithm's declination starts to stabilize developing a recurrent pattern.

Using this declination, the average Euclidean distance from the maximum is 462.4 pixels.

`Linear, random` declination seems to be the one which performs better. Its average Euclidean distance from the maximum is 201.6 pixels and its standard deviation is the lowest of the set with a value of 15.4. This means that it has a relatively good stability. Looking at plot (a) we can easily prove what just said. The current declination starts from a point relatively near to the maximum and never departs from its developing a recurrent pattern.

Looking at `parametric, not random` declination of `SARSA(`λ`)` algorithm we note an high instability. In the very first epochs the distance between the starting point and the maximum slowly decreases, but starting from epoch thirty it starts to messily increase achieving a level near to 350 pixels. This high instability is reflected also in the high level of standard deviation (table 5.5).

`Parametric, random` declination combines instability and inaccuracy. It starts from a distance of about 300 pixels and messily increase its. Its performance is the worst one with an average Euclidean distance of 402.99 pixels and a standard deviation of 66.6 (table 5.5).

Generally speaking we can say that these bad performance are the result of an inadequate training time and of a particular function's shape. In addition to this, as already explained, parametric declinations do worst than corresponding ones because of the reduced amount of movement at each epoch. The relatively better performance of `linear, random` declination depends on the possibility to select a starting point closer to the maximum.

Beale Function	**Mean-ED**	**Standard deviation - ED**
Linear not Random	462.4	25.7
Linear Random	201.6	15.4
Parametric not Random	329.8	62.7
Parametric Random	402.99	66.6

Table 5.5: Euclidean Metric's Performances

Looking at plot (b) of figure 5.3 we note an inconsistency with what said until now. Only `linear, random` configuration makes an highly swinging performance varying between a percentage of value function' s proximity of about 75% to a percentage of 30%. All other declinations makes performances around 100% of value function' s proximity. Even looking at table 5.6 this impression is proved by the high standard deviation of `linear,`

`random` configuration and by the high average of value function' s proximity of all other declinations. However we must not allow ourselves to be deceived. These apparently good performances depends on the function' s shape that is very flat.

Beale Function	Mean- P	Standard deviation - P
Linear not Random	97.25	0.70
Linear Random	54.26	13.89
Parametric not Random	99.06	0.81
Parametric Random	98.88	0.798

Table 5.6: Value Function' s Proximity.

Deception can also be revealed by observing plot (c) of figure 5.3. The relatively best performing declination is the `parametric, not random` one which achieves an effectiveness in finding the maximum of about 0.8 out of 1.0. It is also proved by table 5.9. `Parametric, not random` declination achieves point $(-0.77, 1.5599)$ with a value function of 1997.392 out of 2000. The worst performing one is the `linear, not random` declination with a constant *gap value* equals to 0.

Styblinski-Tang Function Plot (a) of figure 5.4 graphically represents performances of the four possible declinations of `SARSA(`λ`)` algorithm obtained considering the *Euclidean distance metric*.

Looking at the `linear, not random` declination's performance, the Euclidean difference between the agent's position and the global maximum is about 200 pixels. Staring from the twentieth epoch it starts to stabilize around an Euclidean distance level from the maximum of about 40 pixels. Note that in less than twenty epochs the Euclidean distance from the maximum is about the 20% of the starting one. The current declination also reveals a great stability. With an average Euclidean distance of 43.63 pixels and a standard deviation of 41.05, the `linear, not random` declination is the best of the set (table 5.7).

A decisively worse performance is given by the `linear, random` declination. It starts with an Euclidean distance from the maximum of about 360 pixels and messily decreases until achieving a level of 180 pixels. The noise just described is also proved by the highest average Euclidean distance and by the highest standard deviation of the set (table 5.7).

Last two, parametric declinations give us for the first time the possibility

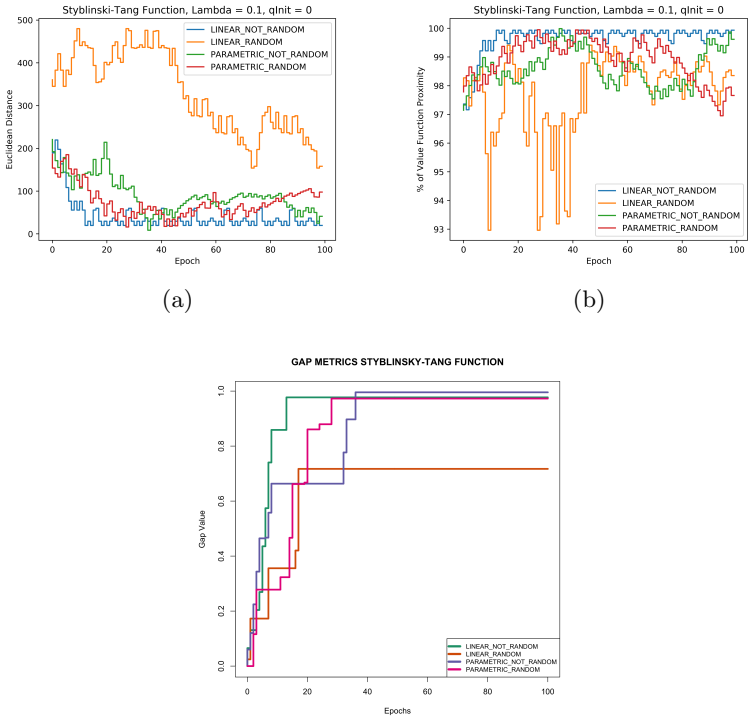


Figure 5.4: Styblinski Function.

to really see behaviour of parametric declination. Starting from about the fiftieth epoch, `parametric, not random` movement starts to imperceptibly increase. This lower amount of movement compared to the non parametric movements depends on the definition of the parametric movement itself. The same behaviour can be seen in `parametric, random` declination starting from epoch 75. This declination performs better then the `parametric, not random` one in the first epochs but it performs worse in the last ones. With an average Euclidean distance from the maximum of 75.26 pixels, and with an average standard deviation of 37.34, the `parametric, random` declinations is generally better than the `parametric, not random` one.

The performance just described is also proved by the percentage of value function' s proximity to the maximum. With an average mean of 99.63% of precision, and a standard deviation of 0.57, the `linear, not random` declination is the best of the set. General good performances are also made

Styblinski-Tang Function	Mean-ED	Standard deviation - ED
Linear not Random	43.63	41.05
Linear Random	330.16	95.39
Parametric not Random	91.79	42.24
Parametric Random	75.26	37.34

Table 5.7: Euclidean Metric's Performances

by all other declinations except for the `linear, random` one. Looking at plot (b) of figure 5.4 we can notice an high instability of this last declination especially until epoch fifty. It than starts to stabilize varying between a proximity level of 97% and 99%.

Styblinski-Tang Function	Mean- P	Standard deviation - P
Linear not Random	99.63	0.57
Linear Random	97.73	1.46
Parametric not Random	98.61	0.66
Parametric Random	98.90	0.74

Table 5.8: Value Function' s Proximity.

Looking at the third plot we notice that the best performance is the one of the `parametric, not random` declination. This result obviously depends on the fact that around the thirty-ninth epoch it reaches the maximum for then decisively distance itself from it. This behaviour is again due to the absence of an adequate training space. This lack prevents the agent from developing an optimal policy.

Table 5.9: Best Soundings

Name	Linear Not Random	Linear Random	Parametric Not Random	Parametric Random
Himmelblau	2486.938 $(2.67, 1.34)$	2498.457 $(3.67, -1.57)$	2485.972 $(-2.1, 3.30)$	2492.11 $(-2.283, 3.234)$
Sphere	3484.44 $(-8.8, -0.67)$	3537.986 $(-3.24, 3.40)$	3557.722 $(-1.034, -1.099)$	3553.626 $(1.69, 1.88)$
Beale	1985.797 $(0, 0)$	1472.184 $(1.69, 2.269)$	1997.392 $(-0.77, 1.559)$	1994.983 $(1.309, -0.89)$
Styblinski-Tang	5153.086 $(-2.67, -2.67)$	5126.192 $(3, -2.97)$	5155.97 $(-2.77, -2.95)$	5154.053 $(-3.17, -2.99)$

Chapter 6

Appendix A

One of the most amazing aspect of the work presented in this thesis is the manner in which the goal has been reached. I want to clarify that during the first weeks, it was not easy to understand the real target of the work. This initial uncertainty led us to implement some strange solutions. Even if some of these approaches were not useful for the goal described in the previous chapters of this thesis, although they were very efficient in different contexts. For this reason we will here recall the full history of the current research activity. In so doing, we identify ten different main steps.

First Step In the first attempt to reach the solution of the problem, we allowed the agent to take five different actions :

- Move North
- Move South
- Move West
- Move East
- Dig.

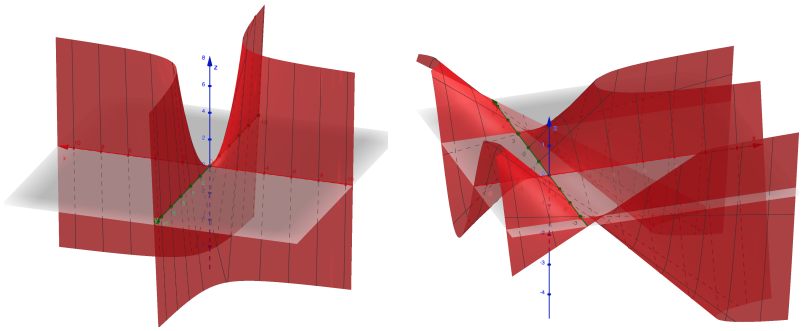
In so doing, we distinguished between the possibility for the agent to take a *movement action* or a *dig action*. In this choice we were inspired by the possibility for humans to make movements for no specific reason. We later understood that this implementation was redundant. Humans move themselves for no apparently reason before decide to take a goal-directed

action. A RL agent *decides* what action taking in the next step immediately after have completed the previous action without needing *time to think*.

In this step the state was represented by three elements :

- Coordinate x
- Coordinate y
- Number of remaining *function soundings*

Unfortunately such a state representation prevents from the possibility to generalize the training. Given two different bivariate functions $f(x, y)$ and $g(x, y)$ and supposing to have trained the RL agent using function $f(x, y)$, when we decide to sound $g(x, y)$ on point x and y already sounded in $f(x, y)$, the result could be different (figure 6.1).



(a) $f(x, y) = 2xy, \quad f(1, 1) = 2$ (b) $g(x, y) = xsin(y), \quad g(1, 1) = 0.017$

Figure 6.1: Different value functions for different functions.

In addition to this, we soon understood that the information concerning the number of remaining function' s soundings is redundant. It did not add any information to our knowledge. In the implementation described above, the agent was in a terminal state when the number of remaining function soundings was equal to zero.

Second Step The second step was globally equal to the first one except for two fundamental aspects. In the current step we introduced the possibility for the agent to make movements of different sizes. This possibility allowed its to reach the goal with a lower training time.

We also redefined the reward function. It was equal to the percentage of proximity to the maximum of the function. Soon we understood that a so defined reward function was completely wrong. In a context of black-box optimization the maximum of the function is unknown and we are unable to compute the percentage of proximity to the maximum.

Third Step In the third step we made an additionally change to the previous one. At the end of each episode, the agent was artificially forced to restart from the coordinates of the previous episode's best rewarded epoch. Results given by the combination of the last two approaches resulted to be very fruitful in the context well-known function optimization but not in the context described in this thesis. Indeed we are forcing the agent to take a specific action, breaking, in so doing, for at least one time the concept of environment formalized through a Markov Decision Process.

Fourth Step In the fourth step we improved the performance obtained through changes made in the second and third step. We succeeded in so doing, imposing the agent to start each episode from the best rewarded epoch from all previous episodes.

Fifth Step The fifth step is one of the most important because of innovations introduced starting from its. The first change made regards actions. In this step we introduced the possibility for the agent to directly choose the size of the movement. There are still four different possible *movement actions* :

- Move North
- Move South
- Move West
- Move East

but the agent could select to move itself in one direction of :

- *movement amount*;
- *movement amount* $\times 2$;
- *movement amount* $\times 3$.

Remember that the unit of measurement of *movement amount* is the *pixel*. The basic amount of movement is 40 pixels. The real movement amount over function is computed through algorithm 12 in Chapter 3. It was proved that using this set of *action movements* the algorithm converged more quickly to the maximum of the function but the precision was lower.

In addition to this, we have also removed the possibility for the agent to distinguish between *move action* and *dig action*. Each time the agent made a movement it also sounded the function obtaining a reward. The reward was positive if its proximity to the global maximum was in percentage $\geq 75\%$, negative in other cases. As already explained, the disadvantage of the last choice is that in the context described in this thesis the objective function is unknown so we are not able to compute the percentage of proximity to the maximum.

Sixth Step In this step we introduced the concept of *delta*. *delta* was computed as the difference between the value of the function sounded at time t and the maximum value sounded until time t.

In this step we also tried to reach an higher control about *exploration* and *exploitation*. In order to do this we introduced a decreasing ϵ factor between episodes. We tried different decreasing functions and we noticed that the best one was the logarithmic one.

In this step the reward was still based on the concept of percentage with an engagement of the *delta*. To be more specific if the *delta* was ≤ 0 or the percentage was ≤ 78 the reward was negative, otherwise it was positive.

Seventh Step In this step we improved the definition of the reward function. We removed the concept of percentage of proximity to maximum and we introduced a reward based exclusively on the difference between the sounded value function at time t and the maximum until time t. This was a little but fundamental adjustment because it removed a relevant mistake from the project.

Eighth Step We can define this step as an experimental step. Here we focused on the possibility to make the agent able to explore more. In order to do this we introduced the possibility to start each episode from a random point. Through this experiment we proved that more exploration gave the possibility to prevent many limit cases but it did not always entail an improvement in algorithm performances in terms of time and precision.

Ninth Step The current one is the second revolutionary step. It introduces two innovative elements: the *parametric movement* and the *stop action*. As explained in chapter 3, the parametric movement allows to compute the *real agent's movement amount* keeping in consideration an approximation of function' s shape.

Speaking about the second innovation introduced, we have to say that when *stop action* was selected, the simulation of the current episode was interrupted. Consequences from introduction of this new action were very interesting. Given a inadequate training space, the *stop action* led to an immediate stop. This depended on the fact that rather than receiving more negative rewards before reaching a maximum better than the current one (i.e. the starting point itself), it computed that the optimal policy consisted in staying around the starting point itself, still moving according to a recurring pattern that minimizes the negative rewards.

Tenth Step This represents the last step before the final version of the project. In this version the state was revolutionised. It was now composed of the last three angles of improvements computed as described in chapter 3. Angles varied between $-\pi$ and $+\pi$ because also a worsening is here considered. Angles were rounded to the nearest multiple of two. The choices introduced in this step were unexpectedly not fruitful. The failure depended on two factors. Last three angles as the unique component of the state were not enough and the rounding cannot be chosen a priori. The correct choice of rounding should depend on the *Lipschitz Constant*.

Let's consider a single-variable function $f(x)$ for x inside a domain D. The magnitude of the slope of $f(x)$ for two points $(x_1, f(x_1))$ and $(x_2, f(x_2))$ is thus

$$|\frac{f(x_1) - f(x_2)}{x_1 - x_2}|. \tag{6.1}$$

A non-negative real number L, which is the smallest upper bound of the slope is called the *Lipschitz Constant* :

$$L = |\frac{f(x_1) - f(x_2)}{x_1 - x_2}| = sup|\frac{df}{dx}|. \tag{6.2}$$

The Lipschitz Constant limits how fast the function can change. If there is no Lipschitz Constant, the function can change extremely fast without border, that is, the function becomes discontinuous[Lip].

For multivariate function the *Lipschitz Constant* is defined as:

$$L_{\mathrm{x_n}} = sup|\frac{\partial f}{\partial x_n}|. \tag{6.3}$$

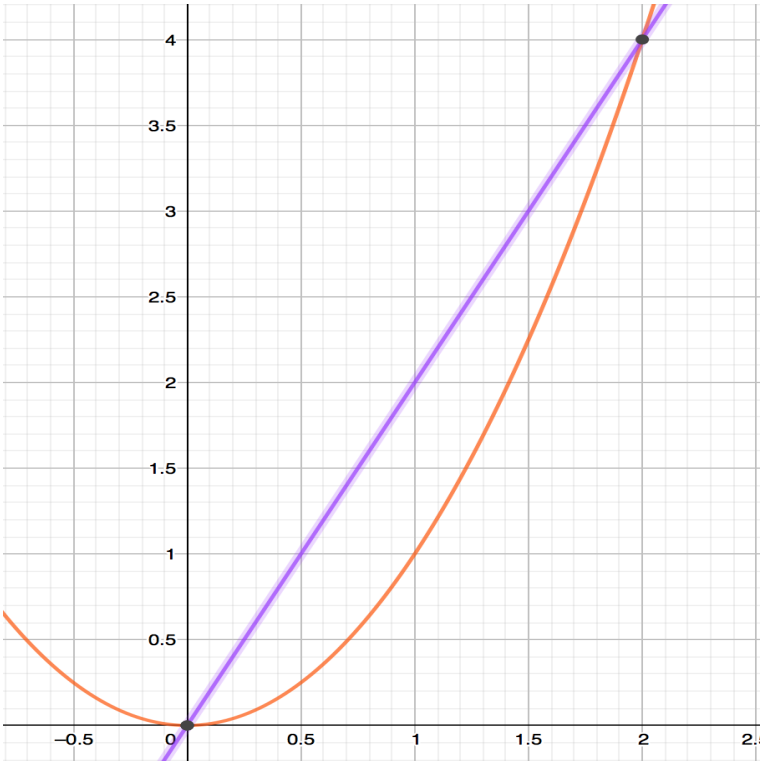


Figure 6.2: Let's consider a $1d$-function $f(x) = x^2$ defined in the domain $x \in [0, 2]$. If $|\frac{df}{dx} f(x)| = 2x$, then $L = \sup |\frac{df}{dx}| = 2 \times 2 = 4$.

Bibliography

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

[Bay] `https://towardsdatascience.com/shallow-understanding-on-bayesian-optimization-324b6c1f7083`. Accessed: 2018-05-20.

[BUR] `http://burlap.cs.brown.edu/tutorials/bd/p2.html`. Accessed: 2018-05-20.

[HBdF11] Matthew Hoffman, Eric Brochu, and Nando de Freitas. Portfolio allocation for bayesian optimization. In *Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence*, UAI'11, pages 327–336, Arlington, Virginia, United States, 2011. AUAI Press.

[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.

[Lip] `https://www.eee.hku.hk/~msang/Numerical_Lipschitz.pdf`. Accessed: 2018-06-05.

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

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

[MND] `https://en.wikipedia.org/wiki/Multivariate_normal_distribution/media/File:Multivariate_Gaussian.png`. Accessed: 2018-05-20.

[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.

[SLA12] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 2951–2959. Curran Associates, Inc., 2012.

[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.