GRADIENT-BASED REINFORCEMENT LEARNING TECHNIQUES FOR UNDERWATER ROBOTICS BEHAVIOR LEARNING

ANDRES EL-FAKDI

PhD Thesis in Computer Engineering

University of Girona
Computer Engineering Department
Computer Vision and Robotics Group (VICOROB)

December 2010

SUPERVISOR:
Marc Carreras

LOCATION:
Girona

SUBMITTED:
December 2010

Andres El-Fakdi: *Gradient-based Reinforcement Learning techniques for underwater robotics behavior learning,* PhD Thesis in Computer Engineering, © December 2010

2 STATE OF THE ART

A commonly used methodology in robot learning is Reinforcement Learning (RL) [152]. In RL, an agent tries to maximize a scalar evaluation (reward or punishment) obtained as a result of its interaction with the environment. This chapter reviews the field of RL. Once the main aspects are described, representative solutions to solving the RLP are presented, with special interest on those able to deal with real-world robots. Two kind of algorithms, Value Function (VF) algorithms and Policy Gradient (PG) algorithms, are analyzed and compared. The main advantages and drawbacks of each approach are described. Finally, a particular combination of both methods which exploits the advantages of VF and PG, called Actor-Critic (AC) algorithms, demonstrate the best performance for solving real robotic tasks. Practical applications of all theoretical algorithms are discussed at the end of this chapter, with conclusions and ideas extracted from them in order to design learning methodologies for underwater robots that represent the main contribution of this dissertation.

2.1 THE REINFORCEMENT LEARNING PROBLEM

The goal of an RL system is to find an optimal policy to map the state of the environment to an action which in turn will maximize the accumulated future rewards. The robot is not told what to do, but instead must discover actions which yield the most reward. Therefore, this class of learning is suitable for *online* robot learning. The agent interacts with a new undiscovered environment selecting the actions computed as the best for each state and receiving a numerical reward for every decision. These rewards will be "rich" for good actions and "poor" for bad actions. The rewards are used to teach the agent and in the end the robot learns which action it must take at each state, achieving an optimal or sub-optimal policy (state-action mapping).

We find different elements when working with RL algorithms. The first is the *agent or learner* which interacts with the *environment*. The environment includes everything that is outside the agent. If we describe the interaction process step by step, the agent first observes the *state* of the environment and, as a result, the agent generates an action and the environment responds to it with a new state and a *reward*. The reward is a scalar value generated by a *reinforcement function* which evaluates the action taken related to the current state. The agent-environment relationship can be seen in Figure 1. Observing the diagram, the interaction sequence can be described: for each iteration step *t*, the agent observes the state s_t and receives a reward r_t. According to these inputs and the RL policy followed at that moment, the agent generates an action a_t. Consequently, the environment reacts to this action changing to a new state s_{t+1} and giving a new reward r_{t+1}. A sequence of states, actions and reward is shown in Figure 2. The most important features of the agent and environment are listed as follows:

AGENT :

- Performs learning and decides on actions.
- Input: the state s_t and reward r_t (numerical value).
- Output: an action a_t.
- Goal: to maximize the future rewards $\sum_{i=t+1}^{\infty} r_i$.

ENVIRONMENT :

- Everything outside the agent.
- Reacts to actions by transferring to a new state.
- Contains the reward function which generates the rewards.

As stated before, the agent interacts with the environment in order to find correct actions. The action applied depends on the current state so, at the end of the learning process, the agent has a *mapping function* which relates every state with the best action taken. To get the best actions at every state, two common processes in RL are used: *exploitation and exploration*. Exploitation means that the agent always selects what is thought to be the best action at every current state. However, exploration is sometimes required to investigate the effectiveness of actions that have not been tried

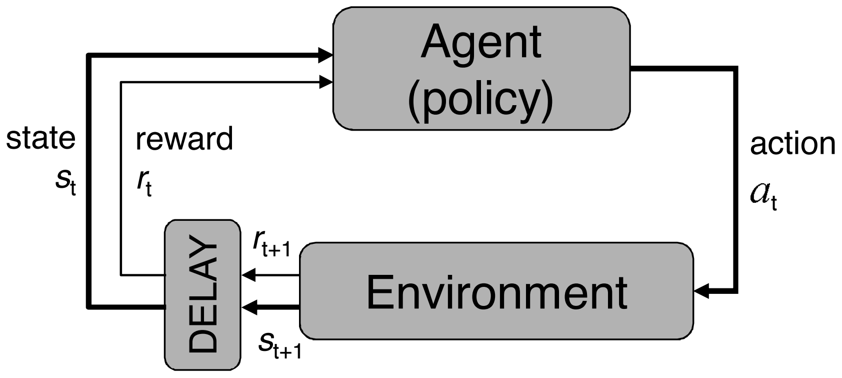

Figure 1: Diagram of the interaction Agent vs. Environment.

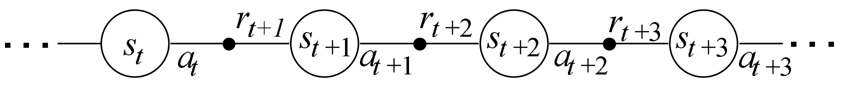

Figure 2: Schematic sequence of states, actions and rewards.

at the current state. Once the learning period is over, the agent does not select the action that maximizes the immediate reward but the one that maximizes the sum of future rewards.

If we take a deeper look, beyond the environment and the agent, four main sub-elements of a reinforcement learning system can be identified: a *policy function*, a *reinforcement function*, a *value function* and a *model* of the environment. These functions define the RLP.

POLICY FUNCTION. This function defines the action to be taken by the agent at a particular state. The policy function represents a mapping between states and actions. In general, policies may be *stochastic*, *deterministic* or *random*. Stochastic policies are represented by $a \sim \pi(a|s)$, where the probability of choosing action *a* from state *s* is contained. Deterministic policies $a = \pi(s)$, also know as *greedy* policies, contain the best mapping known by the agent, that is, the actions supposed to best solve the RLP. A *greedy action* is an action taken from a greedy policy. Finally, a policy can be *random*, in which case the action will be chosen randomly. An $\epsilon - greedy$ policy selects random actions with a probability ϵ and greedy actions with a probability (1 - ϵ). The policy that gets maximum accumulated rewards is called optimal policy and is represented as π^*.

REWARD FUNCTION. This function defines the goal in a reinforcement learning problem. It is located in the environment and, roughly speaking, maps each perceived state of the environment to a single number called a *reward*. The main objective of an RL agent is to maximize the total reward it receives in the long run. The reinforcement function gives an immediate evaluation to the agent. High immediate rewards are not the objective of an RL agent but only the total amount of rewards perceived at the end.

VALUE FUNCTION. As stated before, the reinforcement function indicates what is considered "good" in an immediate sense. The value function defines what will be good in the long run starting from a particular state. In other words, the value of a state is the total amount of estimated reward that the agent is going to receive following a particular policy starting from the current state. There are two kinds of value functions. The first is *State-Value* function $V^{\pi}(s)$, which contains the sum of expected rewards starting from state *s* and following a particular policy π. The second is the *Action-Value* function $Q^{\pi}(s, a)$, which contains the sum of rewards in the long run starting in state *s*, taking action *a* and then following policy π. The action-value function $Q^{\pi}(s, a)$ will be equal to the state-value function $V^{\pi}(s)$ for all actions considered greedy with respect to π. Once the agent reaches the optimal value functions, we can obtain the optimal policy from it.

DYNAMICS FUNCTION. Also known as the model of the environment. This describes the behavior of the environment and, given a state and an action, the model of the environment generates the next state and the next reward. This function is usually stochastic and unknown, but the state transitions caused by the dynamics are contained in some way in the value functions.

Most RL algorithms use these functions to solve the RLP. As will be shown in the next sections, the learning process can be performed either over the value function, the state $V^{\pi}(s)$ or the action value function $Q^{\pi}(s,a)$, or over the policy π itself. These algorithms propose a learning update rule to modify the value or the policy function parameters in order to maximize the received rewards. If the algorithm converges after some iterations, an optimal policy π^{*} can be extracted and the RLP is solved.

2.2 FINITE MARKOV DECISION PROCESSES IN RL

In RL, the agent makes its decision according to the environment's information. This information is presented as the environment's *state* and *reward*. In this thesis, by *the state* we mean whatever information is available to the agent. This section discusses what is required of the state signal and what kind of information we should and should not expect it to provide. If the information contained in the state is sufficient for the agent to solve the RLP, it means that the state summarizes all relevant information. In this case, the state will be called *complete* and the process is said to have accomplished the *Markov Property*. An environment that has the Markov property contains in its state all relevant information to predict the next state. Completeness entails that knowledge of past states, measurements or controls carry no additional information to help us predict the future more accurately. At the same time, future rewards do not depend on past states and actions. This property is sometimes referred to as *the independence of path* property because all that matters is in the current state signal; its meaning is independent of the path taken that led up to it. The response at time $t+1$ to an action taken at time t for a non-Markovian system is shown in Equation 2.1. For this case, the response depends on everything that has happened earlier. Hence, the conditional probability of achieving the next state s_{t+1} and obtaining the reward r_{t+1} when taking action a_t and knowing previous states and actions is defined as

$$Pr\{s_{t+1}=s', r_{t+1}=r' | s_t, a_t, r_t, s_{t-1}, a_{t-1}, ..., s_0, a_0\} \quad (2.1)$$

for all s', r and all possible values of the past events represented by: $s_t, a_t, r_t, ..., r_1, s_0, a_0$. On the other hand,if an environment accomplishes the Markov property, the environment's new state and reward, s_{t+1} and r_{t+1}, will depend only on the state/action representation at time t. This statement can be defined mathematically as

$$\Pr\{s_{t+1} = s', r_{t+1} = r' | s_t, a_t\} \tag{2.2}$$

for all s', r, s_t and a_t. A state signal is considered to have the Markov property if, and only if, Equation 2.1 is equal to Equation 2.2 for all states and actions. If an environment accomplishes the Markov property, its one step dynamics allows us to predict the next state and the next reward given the current state and action. The Markov property is important in RL because the decisions taken are assumed to be functions only of the current state. Hence, although the state at each time step may not fully satisfy the Markov property, it will be approximated as a Markov state.

An RLP which accomplishes the Markov property is called a *Markov Decision Process (MDP)*. Moreover, if the state and action spaces are finite, the environment is considered a *Finite Markov Decision Process (FMDP)*. For a particular FMDP, the stochastic dynamics of the environment can be expressed by a *transition probability* function $P^a_{ss'}$. This function represents the probability of reaching state s' from state s if action a is taken as

$$P^a_{ss'} = \Pr\{s_{t+1} = s' | s_t = s, a_t = a\}. \tag{2.3}$$

In the same way, the expected value of the next reward $R^a_{ss'}$ can be obtained. Given any current state s and action a, together with any state s', we have

$$R^a_{ss'} = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\}. \tag{2.4}$$

Both $P^a_{ss'}$ and $R^a_{ss'}$ represent the most important concepts for defining the dynamics of an FMDP. Most RL techniques are based on FMDPs causing finite state and action spaces. The RLP analyzed in this thesis assumes that the environment is an FMDP. Considering only Markov environments is a risky assumption. The non-Markov nature of an environment can manifest itself in many ways, and the algorithm can fail to converge if the environment does not accomplish all its properties [142]. Following this path, the most direct extension of FMDPs that hides a part of the state to the agent is known as Hidden Markov Models (HMMs).

The underlying environment continues to accomplish the Markov property but the information extracted does not seem Markovian to the agent. The analogy of HMMs for control problems are the Partially Observable Finite Markov Decision Processs (POFMDPs) [98]. The algorithms studied in this dissertation are based on POFMDPs because, in practice, it is impossible to get a complete state for any realistic robot system.

2.3 VALUE FUNCTIONS

Most RL algorithms are based on estimating a *value function* VF. The VF, located in the agent, is related to the MDP dynamics. For a particular learned policy π, the state-value function V^{π} or the action-value function Q^{π} can be expressed in terms of $P^{a}_{ss'}$ and $R^{a}_{ss'}$ and, as will be shown, the optimal VF (V^{*} or Q^{*}) can also be determined. Once this function is obtained, the optimal policy π^{*} can be extracted from it. Before reaching these expressions, a new function has to be defined. As stated in Section 2.1, the goal of RL is to maximize the sum of future rewards. A new function R_t is used in the FMDP framework to express this sum as

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \ldots + r_T. \tag{2.5}$$

This sum finishes at time T, when the task that RL is trying to solve finishes. The tasks, having a finite number of steps, are called *episodic tasks*. However, RL is also suitable for solving tasks which do not finish at a certain number of time steps. For example, in a robotic task, the agent may be continually activated. In this case, the tasks are called *continuous tasks* and can run to infinite. To avoid an infinite sum of rewards, the goal of RL is reformulated to the maximization of the *discounted sum* of future rewards. The future rewards are corrected by a discount factor γ as

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}. \tag{2.6}$$

By setting the discount factor between $0 \leqslant \gamma \leqslant 1$, the infinite sum of rewards does not achieve infinite values and, therefore, the RLP can be solved. In addition, the discount factor allows the selection of the number of future rewards to be maximized. For $\gamma = 0$ only the immediate reward is maximized. For $\gamma = 1$ the maximization will take into account the infinite sum of rewards. Finally, for $0 < \gamma < 1$ only a reduced set of future rewards will be maximized.

The two VFs, V^π and Q^π, can be expressed in terms of the expected future reward R_t. In the case of the state VF, the value of a state s under a policy π, denoted $V^\pi(s)$, is the expected discounted sum of rewards when starting in s and following π thereafter. For the action VF, denoted $Q^\pi(s, a)$, the value of taking action a in state s under policy π is the expected discounted sum of rewards when starting in s, applying action a and following π thereafter. Equations 2.7 and 2.8 formally define these two functions as

$$\begin{aligned} V^\pi(s) &= E^\pi\{R_t | s_t = s\} \\ &= E^\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\right\} \end{aligned} \tag{2.7}$$

and

$$\begin{aligned} Q^\pi(s, a) &= E^\pi\{R_t | s_t = s, a_t = a\} \\ &= E^\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\right\}. \end{aligned} \tag{2.8}$$

These equations define the VFs obtained when following a particular policy π. To solve the RLP, the optimal policy π^*, which maximizes the discounted sum of future rewards, has to be found. As the VFs indicate, the expected sum of future rewards for each state or state/action pair, an optimal VF will contain the maximum values. Therefore, from all the policies π, the one having a VF (V^π or Q^π) with maximum values in all the states or state/action pairs will be an optimal policy π^*. It is possible to have several policies ($\pi_1^*, \pi_2^*, \ldots$) which fulfill this requirement, but only one optimal VF can be found (V^* or Q^*), i. e.,

$$V^*(s) = \max_\pi V^\pi(s) \tag{2.9}$$

and

$$Q^*(s, a) = \max_\pi Q^\pi(s, a). \tag{2.10}$$

In order to find these optimal VFs, the Bellman equation [24] can be employed. This equation relates the value of a particular state or state/action pair with the value of the next state or state/action pair. To relate the two environment states, the dynamics of the FMDP ($P^a_{ss'}$ and $R^a_{ss'}$) is used. The *Bellman optimality equations* for the state and action VFs are given as

$$V^*(s) = \max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^*(s')], \tag{2.11}$$

and

$$Q^*(s,a) = \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma \max_{a'} Q^*(s',a')]. \tag{2.12}$$

The Bellman optimality equations offer a solution to the RLP by finding the optimal VFs V^* and Q^*. Although this solution only works for very small systems and not at all for most continuous systems if the dynamics of the environment is known, a system of equations with N equations and N unknowns can be written using Equation 2.11, N being the number of states. This nonlinear system can be solved resulting in the V^* function. Similarly, the Q^* function can be found.

Once V^* or Q^* is known, the optimal policy can be easily extracted. For each state s, any action a which causes the environment to achieve a state s' with maximum state value with respect to the other achievable states can be considered as an *optimal action*. The set of all the states with their corresponding optimal actions constitutes an optimal policy π^*. It is important to note that to find each optimal action, it is only necessary to compare the state value of the next achievable states. This is due to the fact that the state-value function V^* already contains the expected discounted sum of rewards for these states. In the case where Q^* is known, the extraction of the optimal policy π^* is even easier. For each state s, the optimal action will be the action a, which has a maximum $Q^*(s,a)$ value, i. e.,

$$\pi^*(s) = \arg \max_{a \in A(s)} Q^*(s,a). \tag{2.13}$$

This section has formulated the RLP using a FMDP model. A definition of optimal VFs and optimal policies has been given and the solution to the RLP when the dynamics of the environment is known has been detailed but this rarely happens in real robotics. Even if the model of a real environment is accurate enough, the computational power needed to deal with huge continuous space states and achieve good solutions in real tasks is too high and often unaffordable. Hence, real RLPs lead us to settle for approximations. The following section will describe different methodologies for finding good solutions to the RLP in real robotic scenarios.

2.4 SOLUTION METHODS FOR THE RLP

There are three fundamental methodologies to solve the RLP formulated as an FMDP. This section summarizes the main features of: Dynamic Programming (DP), Monte Carlo (MC) methods and TD learning. Each class has its strengths and weaknesses, which will be discussed in the following.

DYNAMIC PROGRAMMING. This methodology is able to compute the optimal policy π^* given a perfect model of the environment dynamics as an FMDP ($P^a_{ss'}$ and $R^a_{ss'}$). DP algorithms act iteratively to solve the Bellman optimality equations. DP algorithms are able to learn online, that is, while the agent is interacting with the environment in a step-by-step sense. At each iteration, they update the value of the current state based on the values of all possible successor states. The knowledge of the dynamics is used to predict the probability of the next state to occur. As the learning is performed online, the agent is able to learn the optimal policy while it is interacting with the environment. The general update equation for the state VF is given by

$$V_{k+1}(s) = \sum_a \pi(a|s) \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V_k(s')]. \quad (2.14)$$

In this equation, the state VF at iteration $k+1$ is updated with the state VF at iteration k. Most DP algorithms compute the VFs for a given policy. Once this VF is obtained, they improve the policy based on it. Iteratively, DP algorithms are able to find the optimal policy which solves the RLP. The algorithm replaces the old value of s with a new value obtained from the successor states of s and the immediate reward expected from all one-step transitions. Hence, at each iteration step, a *full backup* of all state values is performed. The computational expenses of this process are prohibitive and, together with the perfect model requirements, represent the main drawback of DP in real robotic problems.

MONTE CARLO METHODS. MC do not need a model of the environment. Instead, they use the experience with the environment, sequences of states, actions and rewards, to learn the VFs. MC algorithms interact with the environment following a particular policy π. When the episode finishes, they update the value of all the visited states based on the rewards received. By repeating the learning over several episodes, the VF for a particular policy can be found. MC methods are incremental in an episode-by-episode sense, but not in a step-by-step sense, restricting them from some particular online applications. Equation 2.15 shows the general update rule to estimate the state-value function. At the end of an episode, the current prediction of the state-value $V_k^{\pi}(s)$ is modified according to the sum of rewards R_t received along the whole episode, not the immediate reward expected as DP algorithms do. There is also a learning rate α which averages the values obtained in different episodes. Hence, we have

$$V_{k+1}^{\pi}(s_t) = V_k^{\pi}(s_t) + \alpha[R_t - V_k^{\pi}(s_t)]. \tag{2.15}$$

After the evaluation of a policy, MC methods improve this policy based on the VF learnt. By repeating the evaluation and improving phases, an optimal policy can be achieved. The main advantage of MC methods over DP algorithms is that they do not need a model of the environment. As a drawback, MC methods are not suitable for most continuing tasks, as they can not update the VF until a terminal state is found.

TEMPORAL DIFFERENCE LEARNING. The advantages of the previous methods can be found in TD algorithms. Like DP, TD learning is able to update value estimates at every iteration step based in part on other estimates, without waiting for the final outcome (they *bootstrap*).

Also, similar to MC methods, they do not need the dynamics of the environment but the experience with the environment itself. The general update rule for the state-value function can be seen in Equation 2.16. Similar to MC methods, the updating of the state value $V^{\pi}_{k+1}(s_t)$ is accomplished by comparing its current value with the discounted sum of future rewards. However, in TD algorithms this sum is estimated with the immediate reward r_{t+1} plus the discounted value of the next state. Also, the update rule does not require the dynamics transition probabilities which allow TD algorithms to learn in an unknown environment.

$$V^{\pi}_{k+1}(s_t) = V^{\pi}_{k}(s_t) + \alpha[r_{t+1} + \gamma V^{\pi}_{k}(s_{t+1}) - V^{\pi}_{k}(s_t)]. \tag{2.16}$$

Although TD methods are mathematically complex to analyze, they do not require the dynamics to be known and are fully incremental, allowing them to be performed on-line. These reasons make TD methods the most suitable algorithms for solving the RLP for most robotic tasks, like the ones that will be solved in this dissertation. Therefore, the algorithms studied in this thesis will be based on TD methods.

When dealing with real-world robots, the dominant approach has been to apply different TD-based algorithms to learn a particular VF and then extract the optimal policy from it. These algorithms are called *Value Function algorithms*. The next section describes the mathematical foundation of this methodology, always focusing on those algorithms able to deal with real robotic tasks.

2.5 VALUE FUNCTION ALGORITHMS

Value Function (VF) methodologies find the optimal policy by first searching for the optimal VF and then deducing the optimal policy from the optimal VF. Figure 3 represents a block diagram which is usually followed by RL algorithms based on the VF approach. At every iteration step, the algorithm proposes a learning update rule to modify the current VF $V^{\pi}(s)$ or $Q^{\pi}(s,a)$ and then extracts a policy π to be followed by the agent. If the RL algorithm converges after some iterations, the VF changes to the optimal VF, $V^*(s)$ or $Q^*(s,a)$, from which the optimal policy π^* can be extracted. The solution of the RLP is accomplished by following the state-action mapping contained in the optimal policy π^*.

VF-based methodologies under Markov environments have been a platform for various RL algorithms. The next sections introduce the most representative algorithms using $V^\pi(s)$ and $Q^\pi(s,a)$ respectively.

2.5.1 *Temporal Difference (TD)(λ)*

The TD(λ) [152] is a common example of a state VF approach. The λ term refers to the use of a new concept, the *eligibility trace*. Eligibility traces are a basic tool used by several reinforcement learning algorithms that have been widely used to handle delayed rewards. They can be defined as a bridge between TD methods and MC techniques. As described in Section 2.4, TD methods do not need a model of the environment and update VF using immediate rewards r_{t+1} and value estimates. Similarly, MC methods do not need a model of the environment but they must wait till the episode ends and the total reward R_t is perceived to update the state values. Eligibility trace works as a *decay factor* which introduces a memory into our reward assignment as a temporary record of the occurrence of a particular event stored in a new variable associated to each state called $e_t(s)$ and defined as

$$e_t(s) = \begin{cases} \gamma\lambda\, e_{t-1}(s) & \text{if } s \neq s_t, \\ \gamma\lambda\, e_{t-1}(s) + 1 & \text{if } s = s_t. \end{cases} \tag{2.17}$$

Here, γ is a discount factor and λ is the decay factor defined at the beginning of this section. The range of the decay factor λ is $0 \leqslant \lambda \leqslant 1$. If $\lambda = 0$ all the credit given to previous states is zero except for the last one corresponding to s_t and the TD(λ) reduces to a simple TD method, called TD(0). If $\lambda = 1$ previous visits have full credit. Their value only decreases due to the discount factor γ and our algorithm behaves as an MC method. The TD(λ) algorithm update rule is detailed in Algorithm 1.

At any time, eligibility traces record which states have recently been visited, where "recently" is defined in terms of $\gamma\lambda$. Traces indicate the degree to which each state is *eligible*

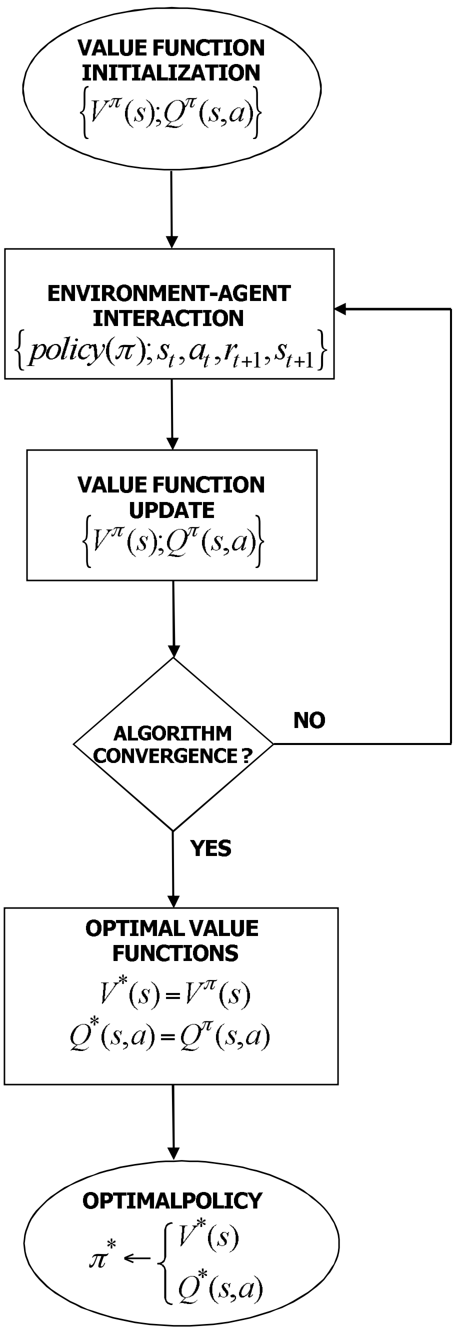
VALUE FUNCTION INITIALIZATION
$\{V^{\pi}(s); Q^{\pi}(s,a)\}$
ENVIRONMENT-AGENT INTERACTION
$\{policy(\pi); s_t, a_t, r_{t+1}, s_{t+1}\}$
VALUE FUNCTION UPDATE
$\{V^{\pi}(s); Q^{\pi}(s,a)\}$
ALGORITHM CONVERGENCE ?
NO
YES
OPTIMAL VALUE FUNCTIONS
$V^*(s) = V^{\pi}(s)$
$Q^*(s,a) = Q^{\pi}(s,a)$
OPTIMALPOLICY
$\pi^* \leftarrow \begin{cases} V^*(s) \\ Q^*(s,a) \end{cases}$

Figure 3: Typical phase diagram of a VF-based RL algorithm, where the VF is updated according to the algorithm. Once the optimal VF is found, the optimal state-action policy π^* is extracted.

Algorithm 1: TD(λ) algorithm.

```
Initialize $V(s)$ arbitrarily and $e(s) = 0$ for all $s \in S$
Repeat until $V(s)$ is optimal:
  a) $s_t \leftarrow$ the current state
  b) $a_t \leftarrow$ action given by $\pi$ for $s_t$
  c) Take action $a_t$, observe reward $r_{t+1}$ and new state $s_{t+1}$
  d) $\delta = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$
  e) $e(s_t) = e(s_t) + 1$
  For all s:
    f) $V(s_t) = V(s_t) + \alpha\delta e(s_t)$
    g) $e(s_t) = \gamma\lambda e(s_t)$
  h) $s_t = s_{t+1}$
```

for undergoing learning changes. Those reinforcing events are the one-step TD error for the state-value prediction indicated by δ. If we take a look at Equation 2.16 in Section 2.4, the update rule of a simple TD(0) algorithm renews only the value of the current state at each iteration, whereas when introducing an eligibility trace, all the values of recently visited states are updated. The propagation of delayed rewards through recently visited states greatly increases the convergence of the algorithm, achieving an optimal policy in fewer iterations. Another on-policy TD control method, the *Sarsa* algorithm [152], uses essentially the same TD(0) method described above, but learns the action-value function rather than the state-value function. Theorems assuring the convergence of state values under TD(0) also apply to this algorithm for action values.

2.5.2 *Q-Learning (QL)*

The QL algorithm [163] is another TD algorithm. As distinguished from the TD(λ) algorithm, QL uses the action-value function $Q^\pi(s,a)$ to find an optimal policy π^*. The $Q^\pi(s,a)$ function has an advantage over the state-value function $V^\pi(s)$. Once the optimal function $Q^*(s,a)$ has been learnt, the extraction of an optimal policy π^* can be directly performed without the requirement of the environment dynamics. The optimal policy will be composed of a map relating each state s with any action a which maximizes the $Q^*(s,a)$ function.

Another important feature of QL is the *off-policy* learning capability. That is, in order to learn the optimal function Q^*, any policy can be followed. The only condition is that all the state/action pairs must be regularly visited and updated. This feature, together with the simplicity of the algorithm, makes QL very attractive for a lot of applications. In real systems, as in robotics, there are many situations in which not all the actions can be executed. For example, to maintain the safety of a robot, an action cannot be applied if there is any risk of colliding with an obstacle. Therefore, if a supervisor module modifies the actions proposed by the QL algorithm, the algorithm will still converge to the optimal policy.

The QL algorithm is described in Algorithm 2. Following the definition of the action-value function, the $Q^*(s, a)$ value is the discounted sum of future rewards when action a is executed from state s, and the optimal policy π^* is followed afterwards. To estimate this sum, QL uses the received reward r_{t+1} plus the discounted maximum value of the future state s_{t+1}.

The first step of the algorithm is to initialize the values of the Q function for all the states s and actions a randomly. After that, the algorithm starts interacting with the environment in order to learn the Q^* function. In each iteration, the update function needs an initial state s_t, the executed action a_t, the new state s_{t+1} which has been achieved, and the received reward r_{t+1}. The algorithm updates the value of $Q(s_t, a_t)$, comparing its current value with the sum of r_{t+1} and the discounted maximum Q value in s_{t+1}. The error is reduced with a learning rate α and added to $Q(s_t, a_t)$. When the algorithm has converged to the optimal Q^* function, the learning process can be stopped. The parameters of the algorithm are the discount factor γ, the learning rate α and the ϵ parameter for the random actions.

Algorithm 2: QL algorithm.

```
Initialize $Q(s, a)$ arbitrarily
Repeat until Q is optimal:
    a) $s_t \leftarrow$ the current state
    b) choose action $a_{max}$ that maximizes $Q(s_t, a)$ over all $a$
    c) $a_t \leftarrow (\epsilon - greedy)$ action, carry out action $a_{max}$ in the
    world with probability $(1 - \epsilon)$ (exploitation), otherwise apply
    a random action (exploration)
    d) Observe the reward $r_{t+1}$ and the new state $s_{t+1}$
    e) $Q(s_t, a_t) =$
    $Q(s_t, a_t) + \alpha[r_{t+1} + \gamma max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$
    f) $s_t \leftarrow s_{t+1}$
```

At each iteration, the algorithm perceives the state from the environment and receives the reward. After updating the Q value, the algorithm generates the action to be taken. The Q function is represented as a table with a different state in each row and a different action in each column. Both the state space and the action space can contain different variables with different values. As can be observed, the algorithm has to store the past state s_t and the past action a_t.

Other algorithms are able to solve the RLP by learning the policy directly. In such algorithms, the policy is explicitly represented by its own *function approximator*, independent of the VF and is updated according to the gradient of the expected reward with respect to the policy parameters. This approach comprises a wide variety of algorithms called *Policy Gradient algorithms*. The survey that follows is focused on PG algorithms that do not require the dynamics to be known and can be performed on-line.

2.6 POLICY GRADIENT ALGORITHMS

Most of the methods proposed in the reinforcement learning community are not applicable to high-dimensional systems as these methods do not scale beyond systems with more than three or four degrees of freedom and/or cannot deal with parameterized policies [114]. PG methods are a notable exception to this statement. Starting with the work in the early 1990s [25, 55], these methods have been applied to a variety of robot learning problems ranging from simple control tasks [26] to complex learning tasks involving many degrees of freedom [117]. The advantages of PG methods for robotics are numerous. Among the most important are that they have good generalization capabilities which allow them to deal with big state-spaces, that their policy representations can be chosen so that it is meaningful to the task and can incorporate previous domain knowledge and that often fewer parameters are needed in the learning process than in VF-based approaches. Also, there is a variety of different algorithms for PG estimation in the literature which have a rather strong theoretical underpinning. In addition, PG methods can be used model-free and therefore also be applied to robot problems without an in-depth understanding of the problem or mechanics of the robot [115]. Studies have shown that approximating a policy directly

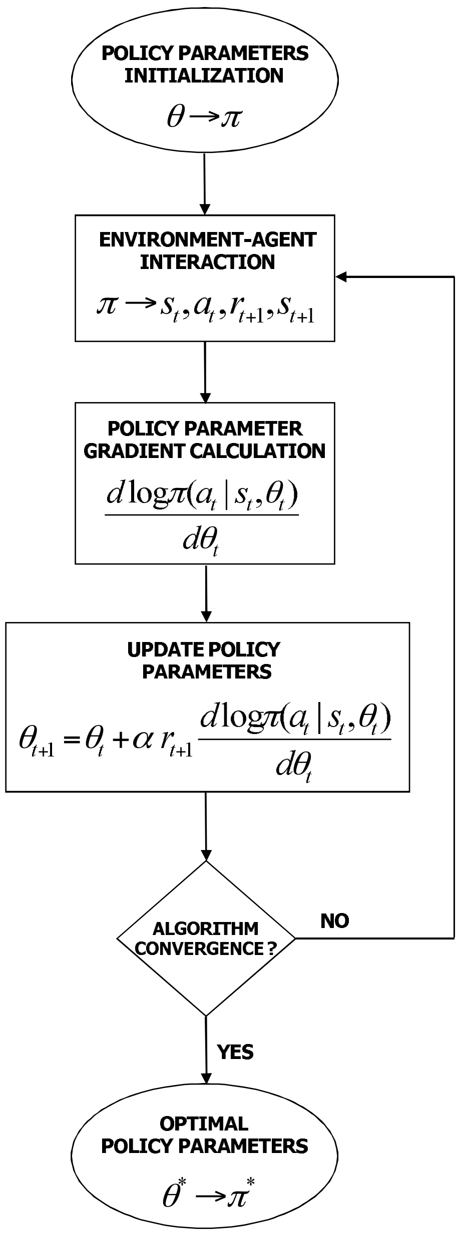
POLICY PARAMETERS INITIALIZATION
$\theta \rightarrow \pi$
ENVIRONMENT-AGENT INTERACTION
$\pi \rightarrow s_t, a_t, r_{t+1}, s_{t+1}$
POLICY PARAMETER GRADIENT CALCULATION
$\frac{d \log \pi(a_t \mid s_t, \theta_t)}{d\theta_t}$
UPDATE POLICY PARAMETERS
$\theta_{t+1} = \theta_t + \alpha\, r_{t+1} \frac{d \log \pi(a_t \mid s_t, \theta_t)}{d\theta_t}$
ALGORITHM CONVERGENCE ?
NO
YES
OPTIMAL POLICY PARAMETERS
$\theta^* \rightarrow \pi^*$

Figure 4: Diagram of a PG-based RL algorithm. The policy parameters are updated according to the gradient of the current policy. Once the optimal policy parameters are found, current policy π becomes optimal π^*.

can be easier than working with VFs [153, 9] and better results can be obtained. Informally, it is intuitively simpler to determine *how to act* instead of *value of acting* [1]. So, rather than approximating a VF, new methodologies approximate a stochastic policy using an independent *function approximator* with its own parameters, trying to maximize the future reward expected. In Equation 2.18 we can see that in the PG approach, the eligibility $e_t(\theta)$ is represented by the gradient of a given stochastic policy $\pi(a_t|s_t, \theta_t)$ with respect to the current parameter vector θ_t as

$$e_t(\theta) = \frac{d \log \pi(a_t|s_t, \theta_t)}{d\theta_t}. \tag{2.18}$$

In section 2.5.1 it is explained that, in the VF approach, the eligibility $e_t(s)$ is bound to the state. It represents a record of the number of visits of the different states while following current policy π. In the PG approach, $e_t(\theta)$ is bound to the policy parameters. It specifies a correlation between the associated policy parameter θ_i and the executed action a_t while following current policy π. Therefore, while the VF's eligibility indicates *how much* a particular state is *eligible* for receiving learning changes, PG's eligibility indicates *how much* a particular parameter of our policy is *eligible* to be improved by learning. The final expression for the policy parameter update is

$$\theta_{t+1} = \theta_t + \alpha r_{t+1} e_t(\theta). \tag{2.19}$$

Here, α is the learning rate of the algorithm and r_t is the immediate reward perceived. Figure 4 represents the phase-diagram of PG-based algorithms. At every iteration step, the algorithm proposes a learning update rule to modify a parameterized policy function $\pi(a|s, \theta)$. This rule is based on the policy derivative with respect to the policy parameters and the immediate reward r_{t+1}. Once the PG algorithm converges, the current policy π becomes the optimal policy π^*. The solution of the RLP is accomplished by following optimal policy π^*.

The next sections introduce several PG methods for learning algorithms. The algorithms presented hereafter detail the most important methodologies from over the last few years and constitute the basic foundation of most successful practical applications which solve the RLP using PG techniques.

2.6.1 *REINFORCE*

The first example of an algorithm optimizing the averaged reward obtained for stochastic policies working with gradient direction estimates is Williams' REINFORCE algorithm [171]. This algorithm learns much more slowly than other RL algorithms which work with a VF and, maybe for this reason, has received little attention. However, the ideas and mathematical concepts presented in REINFORCE were a basic platform for later algorithms. The algorithm operates like the basic PG algorithm presented in Section 2.6 by adjusting policy parameters to a direction that lies along the gradient of expected reinforcement rewards following the expression

$$\Delta\theta_{t+1} = \alpha(r_{t+1} - b_{t+1})e_t(\theta). \tag{2.20}$$

Here, α is the learning rate of the algorithm and $e_t(\theta) = \frac{d \log \pi(a_t|s_t,\theta_t)}{d\theta_t}$ is the eligibility of a particular parameterized stochastic policy $\pi(a_t|s_t, \theta_t)$. As a novelty, Williams introduces a b term called *reward baseline*. This term gives the algorithm a choice to work with immediate ($b = 0$) or delayed ($b \neq 0$) rewards. As can be seen in Equation 2.21, the reward baseline includes a discount factor γ ($0 \leqslant \gamma \leqslant 1$) that maintains an adaptive reward of the upcoming reinforcement based on past experience, i. e.,

$$b_{t+1} = \gamma r_{t+1} + (1-\gamma)b_t. \tag{2.21}$$

Several researchers have previously shown that the use of a reward baseline does not bias the gradient estimate, but motivation to choose a particular baseline form has mainly been based on qualitative arguments and empirical success [148, 170, 41]. The optimal baseline which does not bias the gradient can only be a single number for all trajectories and can also depend on the time-step [117]. However, in the PG theorem it can depend on the current state and, therefore, if a good parameterization for the baseline is known, e.g., in a generalized linear form $b(x_t) = \phi(x_t)^T w$, this can significantly improve the gradient estimation process. However, the selection of the *basis functions* $\phi(x_t)$ can be difficult and often impractical in practice [114] since they may introduce more bias. Therefore, some theorems provide guidance in choosing a baseline [165, 52, 171, 166, 79]. The expression that gives us the update rule for every parameter of our policy is formulated as

$$\theta_{t+1} = \theta_t + \alpha(r_{t+1} - b_{t+1})e_t(\theta). \tag{2.22}$$

As can be easily noticed, the eligibility described in REINFORCE does not accumulate information about past actions, it just represents the correlation between parameters and actions taken at any time step t. Kimura and Kobayashi extended Williams' algorithm to the infinite horizon and modified the eligibility to become an *eligibility trace* [69]. A decay factor λ ($0 \leqslant \lambda \leqslant 1$) similar to the one used in Sutton's TD(λ) is added into the calculation of the new eligibility trace $z_t(\theta)$, i. e.,

$$z_{t+1}(\theta) = \lambda z_t(\theta) + e_t(\theta). \tag{2.23}$$

Hence, we obtain

$$\theta_{t+1} = \theta_t + \alpha(r_{t+1} - b_{t+1})z_{t+1}(\theta). \tag{2.24}$$

Williams' original eligibility $e(\theta)$ contained information about actions immediately executed. Kimura and Kobayashi's eligibility trace $z(\theta)$ acts as a discounted running average of the eligibility. As we move the decay factor λ close to 1, the memory of the agent concerning past actions increases. Adding eligibility traces considerably improves the performance of the algorithm but, as a drawback, this gradient version becomes biased and does not correspond to the correct gradient unlike [54]. The bias-variance tradeoff in gradient estimates are a critical factor in PG algorithms and nowadays it represents a hot topic for most researchers. A good balance between them directly affects the algorithm's performance. The REINFORCE with eligibility traces algorithm is described in Algorithm 3.

The easy structure of these kinds of algorithms, without any computational complexity, and their capability of mildly adapting to POFMDP, make REINFORCE-based methods a good startup for designing algorithms for real robot applications in unknown environments.

Algorithm 3: REINFORCE with eligibility traces algorithm.

```
Initialize parameter vector $\theta_0$ arbitrarily and set $z_0(\theta) = 0$ and $b_0 = 0$.
Repeat until $\pi(a|s, \theta)$ is optimal:
    a) $s_t \leftarrow$ the current state
    b) $a_t \leftarrow$ action given by $\pi(a_t|s_t, \theta_t)$
    c) Take action $a_t$, observe reward $r_{t+1}$
  Calculate eligibility $e_t(\theta)$ and eligibility trace $z_t(\theta)$
    d) $e_t(\theta) = \frac{d \log \pi(a_t|s_t,\theta_t)}{d\theta_t}$
    e) $z_{t+1}(\theta) = \lambda z_t(\theta) + e_t(\theta)$
  Calculate reinforcement baseline and improve policy
    f) $b_{t+1} = \gamma r_{t+1} + (1-\gamma) b_t$
    g) $\theta_{t+1} = \theta_t + \alpha(r_{t+1} - b_{t+1}) z_{t+1}(\theta)$
```

2.6.2 *Gradient Partially Observable Markov Decision Process (GPOMDP)*

The variance of a gradient estimator remains a significant practical problem for PG applications. Although the eligibility traces introduced by Kimura and Kobayashi have proven effective, discounting rewards introduce the bias-variance balance problem: variance in the gradient estimates can be reduced by heavily discounting rewards, but the estimates will be biased. In the same way, the bias can be reduced by not discounting so much, but the variance will then be higher [23]. The GPOMDP algorithm proposed by Baxter and Bartlett aims to improve the gradient estimation. This algorithm demonstrates that a bad selection of a reinforcement baseline b only increases the bias of gradient estimates and does not even improve its variance. In fact, the bias-variance tradeoff can be controlled by just correctly selecting an appropriate value of the decay factor λ. As λ approaches 1, the bias of the estimates decreases, but its variance increases.

Let θ represent the parameter vector of an approximate function $\pi(a|s, \theta)$ that maps a stochastic policy. The gradient function $\frac{d \log \pi(a_t|s_t,\theta_t)}{d\theta_t}$ computes approximations based on a continuous sample path of the Markov chain of the parameterized POFMDP. The accuracy of the approximation is controlled by the decay factor $\lambda \in [0, 1)$ of the eligibility trace $z(\theta)$ which increases or decreases the agent's memory of past actions as in Kimura's algorithm. Furthermore, given a POFMDP and a randomized differentiable policy $\pi(a|s, \theta)$ for all observed states and actions with initial parameter values θ, the update of the eligibility trace $z(\theta)$ and the policy parameter vector θ are completed by

$$z_{t+1}(\theta) = \lambda z_t(\theta) + \frac{d \log \pi(a_t|s_t, \theta_t)}{d\theta_t}, \tag{2.25}$$

and

$$\theta_{t+1} = \theta_t + \alpha r_{t+1} z_{t+1}(\theta). \tag{2.26}$$

Here, α is the learning rate of the algorithm and r is the immediate reward received. As the value of λ increases, the memory of the agent increases, however, variance of the estimates $z_{t+1}(\theta)$ also rises with this parameter. The GPOMDP algorithm is detailed in Algorithm 4.

Algorithm 4: The GPOMDP algorithm.

```
Initialize parameter vector $\theta_0$ arbitrarily and set $z_0(\theta) = 0$.
Repeat until $\pi(a|s, \theta)$ is optimal:
    a) $s_t \leftarrow$ the current state
    b) $a_t \leftarrow$ action given by $\pi(a_t|s_t, \theta_t)$
    c) Take action $a_t$, observe reward $r_{t+1}$
  Calculate eligibility trace $z_{t+1}(\theta)$
    d) $z_{t+1}(\theta) = \lambda z_t(\theta) + \frac{d \log \pi(a_t|s_t, \theta_t)}{d\theta_t}$
  Policy improvement
    e) $\theta_{t+1} = \theta_t + \alpha r_{t+1} z_{t+1}(\theta)$
```

The low mathematical complexity shown by this algorithm together with its online feasibility and the improvements in variance vs. bias tradeoff make the GPOMDP approach a basic algorithm for real applications. At this point, there is no special mathematical difference between the GPOMDP and the REINFORCE approach. Since most of the literature uses the Baxter and Bartlett approach as a basic PG platform for testing, the GPOMDP algorithm was used in some real and simulated testing during the final results of this dissertation.

2.6.3 *Compatible Function Approximation*

As we have previously shown in Section 2.6.1, the natural alternative to using approximate VFs is problematic as these introduce bias in the presence of imperfect basis functions. However, as demonstrated in [73, 153] the term

$Q^{\pi}(x, u) - b^{\pi}(x)$ can be replaced by a *compatible function approximation* $f^{\pi}(s, a)$. The algorithm proposed in [74, 152] defines the VF as a linearly parameterized approximation with its own parameter vector. The main advantage of the proposed algorithm lies in the fact that, due to the small dimension of the parameter vector compared with the space dimension, it is not necessary to find the exact approximated VF, but what they named a reduced *projection* of the VF. Therefore, for the action-value function approach, a compatible function approximation is described by a linear feature-based parameterized approximation without affecting the unbiasedness of the gradient estimate and irrespective of the choice of the baseline $b^{\pi}(s)$ as

$$Q^{\pi}(s, a) - b^{\pi}(s) = f^{\pi}(s, a) = \phi(s, a)^T w. \tag{2.27}$$

Here w represents the parameter vector and $\phi(s, a)$ is the features function. If the algorithm includes an eligibility trace $z(w_t)$, the parameter update at any time step t is performed as

$$\begin{aligned} z_{t+1}(w) &= \lambda z_t(w) + \phi(s_{t+1}, a_{t+1}), \\ w_{t+1} &= w_t + \alpha \delta_t z_{t+1}(w). \end{aligned} \tag{2.28}$$

The variable α denotes the learning rate while λ is the decay factor of its eligibility. The calculation of TD-error δ_t is defined by

$$\delta_{t+1} = r_{t+1} + \phi(s_{t+1}, a_{t+1})^T w_t - \phi(s_t, a_t)^T w_t. \tag{2.29}$$

Similarly to the TD-error, a particular policy $\pi(s, a, \theta)$ can be approximated by

$$\pi(s, a, \theta) = Z_{\theta} \exp(\phi(s, a)^T w_t) \tag{2.30}$$

where $Z_{\theta} = \Sigma_a \exp(\phi(s, a))$, θ is the parameter vector and $\psi(s, a)$ is the features function. The policy updates its parameter vector θ according to the TD-error and the features function $\psi(s_{t+1}, a_{t+1})$ by

$$\theta_{t+1} = \theta_t + \beta \delta_{t+1} \psi(s_{t+1}, a_{t+1}), \tag{2.31}$$

where β denotes a learning rate. The algorithm procedure is summarized in Algorithm 5.

Algorithm 5: Compatible Function Approximation.

```
Initialize policy $\pi(s,a)$ and evaluation VF with parameter
vectors $\theta_0$ and $w_0$ respectively. Set feature vectors $\psi(s,a)$ and
$\phi(s,a)$ for parameterization. Set eligibility trace $z_0(w) = 0$.
Repeat until $\pi(s,a,\theta)$ is optimal:
    a) $a_t \leftarrow$ action given by $\pi(s_t, a_t, \theta_t)$
    b) Take action $a_t$, observe next state $s_{t+1}$ and reward $r_{t+1}$
    Policy evaluation:
    d) $\delta_{t+1} = r_{t+1} + \phi(s_{t+1}, a_{t+1})^T w_t - \phi(s_t, a_t)^T w_t$
    e) $z_{t+1}(w) = \lambda z_t(w) + \phi(s_{t+1}, a_{t+1})$
    f) $w_{t+1} = w_t + \alpha \delta_{t+1} z_{t+1}(w)$
    Policy update:
    g) $\theta_{t+1} = \theta_t + \beta \delta_{t+1} \psi(s_{t+1}, a_{t+1})$
```

2.6.4 *Natural Policy Gradients*

PG algorithms have often exhibited slow convergence. PG convergence speed is directly related to the direction of the gradient, and those found around plateau landscapes of the expected reward may be small and not point towards the optimal solution [114]. These poor performance results made Sham Kakade think whether we were obtaining the right gradient [66]. The GPOMDP approach described in the previous section offered the most straightforward approach of policy improvement, following the gradient in policy parameter space using the steepest gradient ascent,

$$\theta_{t+1} = \theta_t + \alpha \nabla \eta(\theta) \tag{2.32}$$

where $\nabla\eta(\theta)$ is the gradient of the averaged reward function of the POFMDP with the parameter vector θ. In [7], Amari pointed out that the natural PG may be a good alternative to the PG described above. A natural gradient is the one that looks for the steepest ascent with respect to the *Fisher information matrix* [7] instead of the steepest direction in the parameter space. The Fisher information is a way of measuring the amount of information that an observable random variable x carries about an unknown parameter θ upon which the *likelihood function* of θ, $L(\theta) = f(x, \theta)$, depends. The natural gradient of the averaged reward function $\tilde{\nabla}\eta(\theta)$ can be expressed as a function of the Fisher information matrix $F(\theta)$ and the gradient of the averaged reward $\nabla\eta(\theta)$ as

$$\tilde{\nabla}\eta(\theta) = F^{-1}(\theta)\nabla\eta(\theta). \tag{2.33}$$

By inserting a *compatible function approximation* [114] parameterized by the vector w into the PG we obtain

$$\tilde{\nabla}\eta(\theta) = F^{-1}(\theta)G(\theta)w. \tag{2.34}$$

Here the matrix $G(\theta)$ is known as the all-action matrix [66]. It has been demonstrated in [114] that it corresponds to the Fisher information matrix $G(\theta) = F(\theta)$ so the natural gradient can be computed as

$$\tilde{\nabla}\eta(\theta) = F^{-1}(\theta)F(\theta)w = w. \tag{2.35}$$

The policy parameter vector update is finally given by

$$\theta_{t+1} = \theta_t + \alpha w_t. \tag{2.36}$$

Here α is the learning rate of the algorithm. The procedures of a PG algorithm with natural gradient computation is detailed in Algorithm 6.

Algorithm 6: Natural Gradient algorithm.

```
Initialize parameter vector $\theta_0$ arbitrarily and set $z_0(\theta) = 0$ and
$b_0 = 0$.
Repeat until $\pi(s, a, \theta)$ is optimal:
    a) $s_t \leftarrow$ the current state
    b) $a_t \leftarrow$ action given by $\pi(s_t, a_t, \theta_t)$
    c) Take action $a_t$, observe reward $r_{t+1}$
  Calculate the natural gradient of the expected reward $\eta(\theta)$
    d) $\tilde{\nabla}\eta(\theta) = F^{-1}(\theta)F(\theta)w = w_t$
  Policy improvement
    e) $\theta_{t+1} = \theta_t + \alpha w_t$
```

Being easier to estimate than regular PGs, natural gradients are expected to be more efficient and therefore accelerate the convergence process. Also, a more direct path to the optimal solution in parameter space increases convergence speed and avoids premature undesirable convergence.

2.7 VALUE FUNCTION VS POLICY GRADIENT

In previous sections we presented the foundation of VF and PG techniques. Advantages and drawbacks of each one were also discussed when dealing with real robotic tasks. This section compares both methodologies directly with the aim of finding the most suitable solution for the particular experimental tasks where RL techniques are applied in this dissertation.

The dominant approach over the last decade has been to apply RL using the VF approach. As a result, many RL-based control systems have been applied to robotics. In [146], an instance-based learning algorithm was applied to a real robot in a corridor-following task. For the same task, in [59] a hierarchical memory-based RL was proposed, obtaining good results as well. In [35], an underwater robot learns different behaviors using a modified QL algorithm. VF methodologies have worked well in many applications, achieving great success with discrete lookup table parameterization but giving few convergence guarantees when dealing with high dimensional domains due to the lack of *generalization* among continuous variables [152].

RL is usually formulated using FMDPs. This formulation implies a discrete representation of the state and action spaces. However, in some tasks the states and/or the actions are continuous variables. A first solution can be to maintain the same RL algorithms and discretize the continuous variables. If a coarse discretization is applied, the number of states and actions will not be too high and the algorithms will be able to learn. However, in many applications the discretization must be fine in order to assure a good performance. In these cases, the number of states will grow exponentially, making the use of RL impractical. The reason is the high number of iterations necessary to update all the states or state/action pairs until an optimal policy is obtained. This problem is known as the *curse of dimensionality*. In order to solve this problem, most RL applications require the use of generalizing function approximators such as ANNs, instance-based methods or decision-trees. In some cases, QL can fail to converge to a stable policy in the presence of function approximation, even in MDPs, and it may be difficult to calculate $max_{a \in A(s)} Q^*(s, a)$ when dealing with continuous space-states [104]. Another feature of VF methods is that these approaches are oriented to finding deterministic policies. However, stochastic policies can yield considerably higher expected rewards than deterministic ones as in the case of POFMDPs, selecting among different actions with specific probabilities [142]. Furthermore, some problems may appear when the state-space is not completely observable, small changes in the estimated value of an action may or may not cause it to be selected, resulting in convergence problems [27].

Rather than approximating a VF, policy search techniques approximate a policy using an independent function approximator with its own parameters, trying to maximize the future reward expected [115]. The advantages of PG methods over VF-based methods are various. The main advantage is that using a function approximator to represent the policy directly solves the generalization problem. Working this way should represent a decrease in the computational complexity and, for learning systems which operate in the physical world, the reduction in time consumption would be enormous. Furthermore, learning systems should be designed to explicitly account for the resulting violations of the Markov property. Studies have shown that stochastic policy-only methods can obtain better results when working in POFMDPs than those obtained with deterministic value-function methods [142]. In [9] a comparison between a policy-only algorithm [23] and a value QL method [164] is presented where the QL oscillates between the optimal and a suboptimal policy while the policy-only method converges to the optimal policy.

Attempts to apply the PG algorithms to real robotic tasks have shown slow convergence, in part caused by the small gradients around plateau landscapes of the expected return [114]. In order to avoid these situations, studies presented in [7] pointed that the natural PG may be a good alternative to the *typical* PG. Being easier to estimate than regular PGs, they are expected to be more efficient and therefore accelerate the convergence process. Natural gradient algorithms have found a variety of applications over the last few years, as in [125] with traffic-light system optimization and in [161] with gait optimization for robot locomotion.

PG applications share a common drawback, gradient estimators used in these algorithms may have a large variance [88, 74], learning much more slowly than RL algorithms using a VF (see [153]) and they can converge to local optima of the expected reward [96], making them less suitable for on-line learning in real applications. In order to decrease convergence time and avoid local optima, the newest applications combine PG search with VF techniques, adding the best features of both methodologies [20, 74]. These techniques are commonly known as Actor-Critic (AC) methods. In AC algorithms, the critic component maintains a VF, and

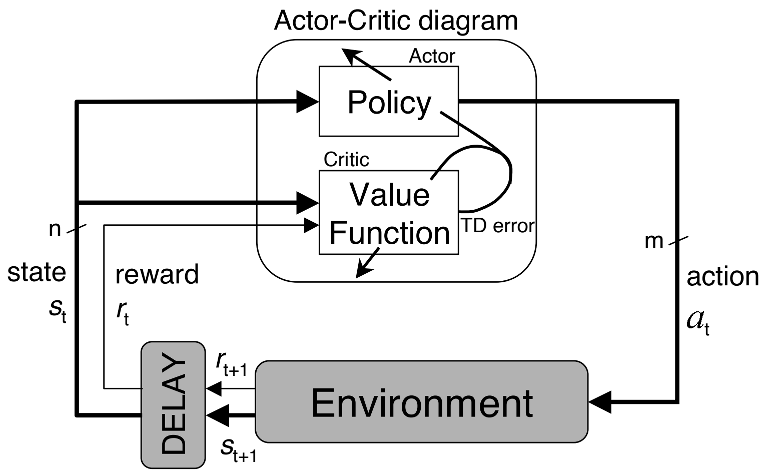

Figure 5: General diagram of the AC methods.

the actor component maintains a separate parameterized stochastic policy from which the actions are drawn. The Actor's PG gives convergence guarantees while the critic's VF reduces variance of the policy update improving the convergence rate of the algorithm [74]. AC basic procedures as well as its most representative algorithms are discussed in the next section.

2.8 ACTOR-CRITIC ALGORITHMS

Actor-Critic (AC) methods [172] combine some advantages of PG algorithms and VF methods. AC methods have two distinctive parts, the *actor* and the *critic*. The actor part contains the policy to be followed. It observes the state of the environment and generates an action according to this policy. On the other hand, the critic observes the evolution of the states and criticizes the actions made by the actor. The critic contains a VF which tries to learn according to the actor policy. A common property of AC methods is the fact that the learning is always *on-policy* which means that, at any time step, the action to be followed is the one indicated by the actor policy. If the agent does not follow this action, the algorithm cannot converge to the RLP solution. The critic will always learn the state-value function for this policy. Figure 5 shows the general diagram of an AC method.

In AC methods, the critic typically uses the state-value function $V^{\pi}(s)$. The actor is initialized to a policy which relates all the states with an action, and the critic learns the state-value function of this policy. According to the values of the visited states, the critic calculates the TD-error and informs the actor. Finally, the actor modifies its policy according to this value difference. If the value of two consecutive states increases, the probability of taking the applied action increases. On the contrary, if the value decreases, the probability of taking that action decreases. AC methods refine the initial policy until the optimal state-value function $V^*(s)$ and an optimal policy π^* are found. The next sections introduce basic AC techniques for training algorithms. The algorithms presented hereafter detail the most important methodologies from over the last few years and constitute the basic foundation of most successful practical applications which solve the RLP using AC techniques.

2.8.1 *Original Actor-Critic*

The main idea and foundation of an AC algorithm was first proposed in [152]. The method follows the structure previously described in Section 2.8. The critic takes the form of a TD error, it learns and critiques whatever policy currently being followed by the actor. Typically, the critic is a state-value function. After each action selection, it evaluates the new state to determine whether things have gone better or worse than expected. That evaluation is the TD error:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t), \tag{2.37}$$

where $V(s)$ is the value function implemented by the critic, r_{t+1} is the immediate reward perceived and $0 \leqslant \gamma < 1$ is the discount factor. The algorithm's procedure is quite simple. The value of the TD error is used to evaluate the actor's current policy, i. e., the action a_t chosen in state s_t. If the value of the TD error is positive, it means that, from state s_t, taking action a_t should be enforced in the future. If the value of the TD error is negative, it means that the tendency to select a_t from s_t should be weakened. Suppose the actor is implemented by a stochastic policy $\pi(a|s, \theta)$ parameterized by vector θ. The critic's error estimate δ_t allows the actor to upgrade its policy according to

$$\theta_{t+1} = \theta_t + \alpha\delta_t \frac{d \log \pi(a_t|s_t, \theta_t)}{d\theta_t}. \tag{2.38}$$

Here α is the learning rate for the actor. The critic updates its VF according a common TD procedure given by

$$V(s_t) = V(s_t) + \beta\delta_t. \tag{2.39}$$

Here the β term corresponds to the learning rate for the critic. Many of the earliest RL systems that used TD methods were AC methods [172, 20]. Since then, the attention given to these methods has increased greatly, mainly because they have two significant apparent advantages: they require minimal computation to select actions and they can learn an explicitly stochastic policy [152]. The algorithm procedure is summarized in Algorithm 7.

Algorithm 7: Original Actor-Critic (AC).

```
Initialize policy $\pi(a|s,\theta)$ with initial parameters $\theta = \theta_0$, its
derivative $d\log\pi(a|s,\theta)$, an arbitrarily VF $V^{\pi}(s)$.
Repeat until $\pi(a|s,\theta)$ is optimal:
    a) $a_t \leftarrow$ action given by $\pi(a_t|s_t,\theta_t)$
    c) Take action $a_t$, observe next state $s_{t+1}$ and reward $r_{t+1}$
    Critic evaluation and update:
      b) $\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$
      e) $V(s_t) = V(s_t) + \alpha\delta_t$
    Actor update:
      d) $\theta_{t+1} = \theta_t + \alpha\delta_t \frac{d\log\pi(a_t|s_t,\theta_t)}{d\theta_t}$
```

2.8.2 *Traced Actor-Critic*

In Section 2.6.1, Kimura and Kobayashi introduced the eligibility trace $z(\theta)$ to the PG estimates proposed in REINFORCE. This addition improved the overall performance of the algorithm considerably. Good results encouraged them to apply trace calculation to AC methods. Kimura and Kobayashi's algorithm [68] proposes a classic AC method which follows the rules previously presented in Figure 5. As will be appreciated in the various AC methodologies discussed in this section, the utilization of the eligibility trace in AC algorithms can vary from one algorithm to another. Some methods use them as a part of the actor update, others use them in the critic and others apply them to both sides. Kimura and Kobayashi's proposed algorithm utilizes the eligibility traces for the actor parameter update. The next lines summarize the whole algorithm's procedure. The actor is implemented by a stochastic policy $\pi(a|s,\theta)$ parameterized by vector θ. The critic attempts to estimate the evaluation function for the current policy. The reinforcement used is the value function TD-error δ as shown in the following equation

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t). \tag{2.40}$$

Here r_{t+1} is the immediate reward perceived and $0 \leqslant \gamma < 1$ is the discount factor. On the actor's side, the eligibility trace $z(\theta)$ is represented as the decayed sum of the eligibility of a stochastic policy $\pi(a|s,\theta)$ with respect to the current parameter vector θ as

$$z_{t+1}(\theta) = \lambda z_t(\theta) + \frac{d \log \pi(a_t|s_t,\theta_t)}{d\theta_t}, \tag{2.41}$$

$0 \leqslant \lambda \leqslant 1$ being the decay factor of the eligibility trace. The critic's error estimate δ allows the actor to upgrade its policy according to

$$\theta_{t+1} = \theta_t + \alpha \delta_t z_{t+1}(\theta). \tag{2.42}$$

Here α is the learning rate for the actor. The critic updates its VF according to a common TD procedure given by

$$V(s_t) = V(s_t) + \beta \delta_t. \tag{2.43}$$

Here the β term corresponds to the learning rate for the critic. The algorithm procedure is summarized in Algorithm 8.

2.8.3 *Single Value and Policy Search (VAPS)*

The VAPS algorithm [18] allows PG and VF methods to be combined simultaneously. With the aid of a mixing term β, the VAPS algorithm computes the differential error value of a state $e_{va-po}(s)$ as a combination of a PG and a VF as

$$e_{va-po}(s_{t+1}) = (1-\beta)\delta_{va}(s_{t+1}) + \beta(b - \gamma r_{t+1}), \tag{2.44}$$

where δ_{va} is the TD-error function for the VF approach and $b - \gamma r_{t+1}$ corresponds to the averaged reward of a PG

Algorithm 8: Traced AC: Eligibility traces for actor update.

```
Initialize policy $\pi(a|s,\theta)$ with initial parameters $\theta = \theta_0$, its
derivative $d\log\pi(a|s,\theta)$, an arbitrarily VF $V^{\pi}(s)$ and set the
eligibility trace $z(\theta) = 0$.
Repeat until $\pi(a|s,\theta)$ is optimal:
    a) $a_t \leftarrow$ action given by $\pi(a_t|s_t,\theta_t)$
    c) Take action $a_t$, observe next state $s_{t+1}$ and reward $r_{t+1}$
    Critic evaluation and update:
      b) $\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$
      e) $V(s_t) = V(s_t) + \alpha\delta_t$
    Actor update:
      c) $z_{t+1}(\theta) = \lambda z_t(\theta) + \frac{d\log\pi(a_t|s_t,\theta_t)}{d\theta_t}$
      d) $\theta_{t+1} = \theta_t + \alpha\delta_t z_{t+1}(\theta)$
```

algorithm which uses a reinforcement baseline b and a discount factor γ, like the classic PG algorithms detailed in Section 2.6.1. Adjustments of parameter β between 0 and 1 allows us to go back and forth between both RL methods. When $\beta = 0$, the algorithm totally learns the action-value function $Q^{\pi}(s,a)$ that satisfies the Bellman equation

$$\delta_{va}(s_{t+1}) = r_{t+1} + \gamma max_{a_{t+1}} Q^{\pi}(s_{t+1}, a_{t+1}) - Q^{\pi}(s_t, a_t). \tag{2.45}$$

On the other hand, if $\beta = 1$, the algorithm directly learns a policy that will minimize the expected total discounted reward. An eligibility trace is added for the stochastic policy update

$$z_{t+1}(\theta) = \lambda z_t(\theta) + \frac{d\log\pi(a_t|s_t,\theta_t)}{d\theta_t}. \tag{2.46}$$

Here λ is the decay factor of the eligibility trace. The current differential error value of a state $e_{va-po}(s)$ is finally used for updating the policy parameters as shown in

$$\Delta\theta_t = -\alpha\left[\frac{de_{va-po}(s_{t+1})}{d\theta} + e_{va-po}(s_{t+1})z_{t+1}(\theta)\right] \tag{2.47}$$

and

$$\theta_{t+1} = \theta_t + \Delta\theta_t. \tag{2.48}$$

The variable α represents a learning rate. The VAPS algorithm procedure is summarized in Algorithm 9. When $\beta = 0$, the algorithm behaves as a pure VF method, the new algorithm converges but it cannot learn the optimal policy as the reward function includes the Bellman residual. When $\beta = 1$, the algorithm behaves as a pure PG method. In this case the algorithm converges to optimality, but slowly since there is no VF catching the results in the long sequence of states near the end. By combining the two approaches, given a particular RLP, this algorithm can offer a much quicker solution than either of the two alone for the same problem. It is important to mention that this algorithm is known to be biased and it does not approximate a true gradient. Therefore, it has no convergence guarantees.

Algorithm 9: VAPS algorithm.

```
Initialize policy $\pi$, an arbitrarily $Q^{\pi}(s,a)$ and set $z(\theta) = 0$.
Repeat until $\pi(a|s,\theta)$ is optimal:
    a) $a_t \leftarrow$ action given by $\pi(a_t|s_t,\theta_t)$
    b) Take action $a_t$, observe next state $s_{t+1}$ and reward $r_{t+1}$
    Evaluate the policy mixing the action value error and the
    expected discounted reward:
    c) $\delta_{va}(s_{t+1}) = r_{t+1} + \gamma Q^{\pi}(s_{t+1}, a_{t+1}) - Q^{\pi}(s_t, a_t)$
    d) $\delta_{va-po}(s_{t+1}) = (1-\beta)\delta_{va}(s_{t+1}) + \beta(b - \gamma r_{t+1})$
    Update eligibility trace and improve policy:
    e) $z_{t+1}(\theta) = \lambda z_t(\theta) + \frac{d \log \pi(a_t|s_t,\theta_t)}{d\theta_t}$
    f) $\theta_{t+1} = \theta_t - \alpha \left[\frac{d e_{va-po}(s_{t+1})}{d\theta} + e_{va-po}(s_{t+1}) z_{t+1}(\theta)\right]$
```

2.8.4 *The Natural Actor-Critic (NAC)*

The NAC algorithm [117] was proposed by Jan Peters and Stefan Schaal. This algorithm combines AC techniques with natural gradient computation, detailed in Section 2.6.4, with the aim of obtaining advantages from both methodologies and achieve faster convergence. Stochastic natural PGs allow actor updates while the critic computes the natural gradient and the VF parameters by linear regression simultaneously. The actor's PG gives convergence guarantees under function approximation and partial observability while the critic's VF reduces variance of the estimates update improving the convergence process. The parameter update procedure starts on the critic's side. At any time step t, the features $\phi(s)$ of the designed basis function are updated according to the gradients of the actor's policy parameters θ as shown in

$$\tilde{\phi}(s_t) = [\phi(s_{t+1}), 0] \tag{2.49}$$

and

$$\hat{\phi}(s_t) = [\phi(s_t), \frac{d \log \pi(a_t|s_t, \theta_t)}{d\theta_t}]. \tag{2.50}$$

These features are then used to update the critic's parameters and find the natural gradient. This algorithm uses a variation of the Least Squares Temporal Difference (LSTD)(λ) [29] technique called LSTD-Q(λ). Thus, instead of performing a gradient descent, the algorithm computes estimates of matrix A and b and then solves the equation $b + A\tau = 0$ where the parameter vector τ encloses both, the critic parameter vector ν and the natural gradient vector w:

$$\begin{aligned} z(s_{t+1}) &= \lambda z(s_t) + \hat{\phi}(s_t), \\ A(s_{t+1}) &= A(s_t) + z(s_{t+1})(\hat{\phi}(s_t) - \gamma\tilde{\phi}(s_t)), \\ b(s_{t+1}) &= b(s_t) + z(s_{t+1})r_t, \\ [\nu_{t+1}, w_{t+1}] &= A^{-1}_{t+1} b_{t+1}. \end{aligned} \tag{2.51}$$

Here, λ is the decay factor of the critic's eligibility, r_t is the immediate reward perceived and γ represents the discount factor of the averaged reward. On the actor's side, the current policy is updated when the angle between two consecutive natural gradients is smaller than a given threshold, $\epsilon \ \angle(w_{t+1}, w_t) \leqslant \epsilon$ according to

$$\theta_{t+1} = \theta_t + \alpha w_{t+1}. \tag{2.52}$$

Here, α is the learning rate of the algorithm. Before starting a new loop, a forgetting factor β is applied to the critic's statistics as shown in

$$\begin{aligned} z(s_{t+1}) &= \beta z(s_{t+1}), \\ A(s_{t+1}) &= \beta A(s_{t+1}), \\ b(s_{t+1}) &= \beta b(s_{t+1}). \end{aligned} \tag{2.53}$$

The algorithm's procedure is summarized in Algorithm 10.

Algorithm 10: NAC algorithm with LSTD-Q(λ)

```
Initialize $\pi(a|s,\theta)$ with initial parameters $\theta = \theta_0$, its derivative
$\frac{d \log \pi(a_t|s_t,\theta_t)}{d\theta_t}$ and basis function $\phi(s)$ for the VF $V^{\pi}(s)$. Draw
initial state $s_0$ and initialize $z = A = b = 0$.
for $t = 0$ to $n$ do:
    Generate control action $a_t$ according to current policy $\pi_t$.
    Observe new state $s_{t+1}$ and the reward obtained $r_t$.
    Critic Evaluation (LSTD-Q($\lambda$))
        Update basis functions
        $\tilde{\phi}(s_t) = [\phi(s_{t+1})^T, 0^T]$
        $\hat{\phi}(s_t) = [\phi(s_t)^T, (\frac{d \log \pi(a_t|s_t,\theta_t)}{d\theta_t})^T]^T$
        Update sufficient statistics:
        $z(s_{t+1}) = \lambda z(s_t) + \hat{\phi}(s_t)$
        $A(s_{t+1}) = A(s_t) + z(s_{t+1})(\hat{\phi}(s_t) - \gamma\tilde{\phi}(s_t))^T$
        $b(s_{t+1}) = b(s_t) + z(s_{t+1})r_t$
        Update critic parameters:
        $[v_{t+1}^T, w_{t+1}^T] = A_{t+1}^{-1} b_{t+1}$
    Actor Update
        If $\angle(w_{t+1}, w_{t-\tau}) \leqslant \epsilon$, then update policy parameters:
        $\theta_{t+1} = \theta_t + \alpha w_{t+1}$
        Forget sufficient statistics:
        $z(s_{t+1}) = \beta z(s_{t+1})$
        $A(s_{t+1}) = \beta A(s_{t+1})$
        $b(s_{t+1}) = \beta b(s_{t+1})$
end
```

2.9 APPLICATIONS OF POLICY GRADIENT METHODS IN ROBOTICS

A background of representative RL techniques has been presented. The survey developed along previous lines started with general solutions to solve the RLP and, as we continued, the interest moved to all those RL methodologies that show interesting advantages when solving real robotic tasks. Fast convergence, on-line capabilities and the power to deal with high dimensional domains are some of the special requirements this survey is looking for. Among all the algorithms reviewed, PG algorithms offer serious advantages for real robotic tasks. With the theoretical aspects clear, this sections aims to discuss practical contributions obtained in real robot learning using PG methods.

2.9.1 *Robot Weightlifting*

If we take a look back in time chronologically, the first successful attempts to apply gradient techniques to real robotics were very simple and barely worried about variance of the estimates, true gradients or convergence issues. Despite the simplicity of those first applications, they kept the essence of PG methods. An example is the algorithm designed by Rosenstein and Barto in 2001 [131]. The proposed method solves the RLP of a weightlifter automata that tries to perform a payload shift episodic task with a limited maximum torque at each of the three robot joints. The policy is represented by a Proportional Derivative (PD) controller whose parameters are the policy parameters to be improved though learning. The update rule is very simple: the algorithm performs a *simple random search* in the parameter space by adding a small amount to each parameter at every iteration step, this amount being normally distributed with zero mean and a variance equal to a particular search size. In order to assure exploration, the algorithm updates the parameters with a fixed probability β or keeps the best ones found at that moment with probability $1-\beta$. The search size is also updated every iteration, modified by a decay factor γ until a minimum is reached. Every time some new set of parameters is obtained, the PD controller tries them in an attempt to lift the weight. Such off-line configuration disconnects the learning algorithm from the real continuous state space and actions, leaving the real interaction to the PD controller. Although the results that Rosenstein and Barto presented were simulated, its weightlifter was able to learn the task, but the number of trials to achieve a good solution was very high.

2.9.2 *Wheeled Mobile Robot*

In looking for procedures to improve performance of RL methods for real tasks, and even though this one is not a PG technique, it is worth discussing here the ideas presented by Smart and Kaelbling around 2002. This algorithm uses VF techniques to solve an RLP were a wheeled mobile robot learns a control policy for corridor following and obstacle avoidance tasks [147]. The policy is extracted from a QL-based method which follows the procedures stated in Section 2.5.2, with the difference that, instead of applying the common tabular representation of $Q(s, a)$, a continuous function approximator for the Q function is used, allowing the algorithm to deal with continuous state-space representation encountered in the real world. The robot has 2 DoF, rotation around the Z axis and X translation forward and backward. The translation DoF is not learnt and the goal of the algorithm is to learn a good policy for controlling the rotation. The learning is formulated as an episodic task and the rewards are computed as a function of the steps taken to reach the goal at every episode. One of the main contributions of this algorithm is that, in order to overcome the long convergence times where simple RL techniques certainly fall, Smart and Kaelbling's proposal includes prior knowledge in the learning system. To do so, the learning procedure is split into two phases. In the first phase of learning, the robot is controlled by a supplied policy. This can either be a human directly controlling the robot or some kind of control code. During this phase, the learning algorithm is passively watching state transitions and updating the Q function. In the second phase, once the Q function is complete enough to adequately assume command, the learned policy takes control of the robot and continues improving itself. Results show that around 30 episodes are needed to achieve an initial policy in phase one and another 35 episodes more to improve it during phase two.

2.9.3 *Gait Optimization for Quadruped*

Following the procedures of Rosenstein and Barto, in 2004 Kohl and Stone applied a simple random search PG algorithm to learn a task where the commercial quadrupedal robot created by Sony, AIBO, tries to find the fastest possible walking speed [72]. The robot's policy is defined by a set of parameters which refer to different aspects of AIBO's dynamics: front and rear locus, robot heights, etc. Tuning them modifies gait behavior and consequently, the walking speed. Starting from an initial parameter vector, the algorithm generates a set of random policies by adding a small amount to each parameter. Each of the randomly generated policies is evaluated and classified by means of an averaging score and the best adjustment vector from all the policies is extracted. In the next trial, the policy is modified with this adjustment and tested. The process is repeated until convergence. The learning algorithm finds the best policy after 3 hours of learning and, like Rosenstein and Barto's, the learning process of the algorithm proposed by Kohl and Stone is quite simple and performed off-line between each trial, sharing the same slowness found in the previous method. The results demonstrated that PG techniques were a promising application to solve the RLP, but improvements must be made to make it viable for real robotic tasks.

2.9.4 *Optimizing Passive Dynamics Walk*

Up to this point, the attempts to solve real RL tasks via PG methods have been rather simple. They were based on tuning an existing controller in open-loop trajectories with its final performance limited by the initial controller design. In 2004, Tedrake proposes a PG technique to improve a passive dynamics controller [155]. The resultant algorithm is an AC based method similar to the one discussed in Section 2.8.2 and is used to solve an RLP were a biped robot learns to walk from a blank-slate. In order to decrease the number of DoFs and actuators, the mechanical design of the robot was based on a passive dynamic walker [94], with a resultant biped of only 9 DoFs and 4 actuators. The actor is represented by a deterministic policy which is updated using discounted eligibilities. The critic provides an estimate of the VF to the actor by observing the robot's performance on a Poincaré map. Both actor and critic are represented by parameterized linear approximators using non-linear features. As input, the policy has the actual value of the 9 DoFs (6 internal and 3 for the robot's orientation). As output, a 4 dimension vector generates the desired torque for the actuators. The results show that the robot begins to walk after only one minute of learning from blank-slate and the learning converges to the desired trajectory in less than 20 minutes. Also, once the policy is learned, on-line learning capabilities allow the robot to adapt to small changes in the terrain easily. As a drawback, 9 DoFs and 4 actuators represent a high dimensional domain for the policy. Increasing the number of dimensions to upgrade the performance may result in scaling problems. As DoFs are added, the reward assignment to each actuator becomes more difficult, requiring more learning trials to obtain a good estimate of the correlation.

2.9.5 *Learning Planar Biped Locomotion*

Along the same lines as Tedrake's walker, in 2005 Matsubara designed an AC algorithm to make a biped robot learn to walk [91]. The AC method follows the lines described in Section 2.8.2. The actor's policy is represented by a Central Pattern Generator (CPG) composed of a neural oscillator [92], whose weights are the policy parameters to be learnt by the actor. The critic is represented by a VF with its own set of parameters, which estimates the value of the state as a function of the biped's maintenance of the upright position as the first requirement and the forward progress in a second term. Both policy and value approximators are modeled using an ANN. Mechanically, Matsubara's robot differs from Tedrake's in the fact that the design of this biped is not based on passive walker dynamics, thus, an incorrect gait will cause the robot to fall down and the learning process finishes without reaching the objective. Therefore, the aim for a fast convergence algorithm was a priority. In order to increase convergence speed, Matsubara proposes to reduce the dimensionality of the continuous state-space of the problem turning the MDP into a POFMDP. To do so, the input state to the actor includes only the DoFs corresponding to the two hip joints and their derivatives forming a 4 dimensional input vector. Therefore, the rest of the states and the state of the neural oscillator remain hidden for the learning algorithm. The reduction of the number of states make the problem easier to solve but, as a drawback, a more precise value of the known states is needed. For this purpose, a good sensory feedback is a must to achieve satisfactory results. Considering a trial as one attempt to walk without falling, results show that an appropriate policy is learned after around 3000 trials have been completed.

2.9.6 *Robot T-ball*

Successful applications of PG algorithms for real robotic tasks point towards a specific class of methods: AC algorithms. AC algorithms join special properties vital for RL algorithms to succeed in a real task. The actor's PG gives convergence guarantees while the critic's VF reduces variance of the policy update, improving the convergence rate of the algorithm. In 2003, Jan Peters and Stefan Schaal developed an AC algorithm to solve an RLP where a robotic arm learns to hit a ball properly so that it flies as far as possible, like a baseball player would try to do [115]. The algorithm uses Dynamic Movement Primitives (DMP) [138] as a compact representation of a movement. As a novelty, the actor's policy is represented by a parameterized function approximator which is updated with respect to the natural gradient estimates instead of the typical ones. The critic is represented by a VF approximation with its own set of parameters. Peters and Schaal called this algorithm NAC. The input state of the actor's policy is given by the arm's joint angles and derivatives corresponding to the 7 DoFs of the arm. As output, the actor's policy generates accelerations for each of the joints. The critic evaluates the rewards by means of a computer vision algorithm which tracks the whole system and evaluates the performance of every trial. This methodology, as with the wheeled mobile robot in Section 2.9.2, benefits from including prior knowledge in the learning system. This time, a human teaches a rudimentary stroke by taking the robotic hands between his and hitting the ball when it comes. With this information, the arm tries to hit the ball without help in subsequent trials. Results show that after approximately 200-300 trials the ball is correctly hit by the robot arm.

2.9.7 *Learning Motor Skill Coordination*

The results obtained in Section 2.9.6 with the combination of learning from demonstration and DMP techniques has inspired more research in this field. In [76], a robot arm acquires new motor skills by learning the couplings across motor control variables. The skills are first trained from demonstration and, in order to reduce the number of states, encoded in a modified version of DMP. In order to learn new values for the coordination matrices, the proposed method uses an Expectation Maximization (EM)-based RL algorithm called Policy Learning by Weighting Exploration with the Returns (PoWER) developed in [71]. The policy parameterization allows the algorithm to learn the couplings across the different motor control variables. One major advantage of this algorithm over other PG-based approaches is that it does not require a learning rate parameter, always a critical parameter to tune in PG algorithms. Also, the proposed algorithm can be combined with importance sampling to make better use of past experience to focus future exploration of the parameters space. Two learning experiments were performed: a reaching task, where the robot needs to adapt the learned movement to avoid an obstacle, and a dynamic pancake-flipping task. The proposed methodology successfully accomplished both tasks, demonstrating the fitness of the proposed approach in such highly-dynamic real-word tasks.

2.9.8 *Traffic Control*

The performance demonstrated by NAC class algorithms encouraged researchers to try this kind of methodology in other practical applications. In 2006, Richter and Aberdeen used an efficient on-line version of the NAC to solve an RLP where an agent tries to learn an optimal traffic signal control for a city [125]. Considering a street intersection as a POFMDP, the actor's policy is defined as a parameterized function approximator while the critic is represented by a VF approximation with its own set of parameters. The actor's state input variables are measurable traffic parameters related to sequences of cars and time cycles. As output, the agent operates the traffic light at the intersection. The critic's VF computes estimates of the averaged reward considering the immediate reward as the number of cars that entered the intersection over the last time step. Even though the results presented were simulated, the nature of the problem makes no substantial difference between the simulation and the real world. The learning algorithm does not depend directly on a model of the traffic flow, just the ability to interact with it, so the final controller learned can be plugged into a more accurate simulation without modification. Results show that the NAC algorithm outperforms existing traffic controllers, becoming a serious candidate for use in real cities.

2.9.9 *Summary*

A summary of the practical applications described in this section is presented in Table 1. Both, theoretical algorithms and practical applications point towards AC methods as the best for real robotics. Also, benefits from introducing initial reliable knowledge into the learning system are great.

Application	**On/Off line**	**Simulated or real?**	**Need a model?**	**Method used**
Robot Weightlifting	Off-line	Simulated	Yes	PG
Wheeled Mobile Robot	On-line	Real	No	VF
Gait Optimization	Off-line	Real	No	PG
Passive Dynamics Walk	On-line	Real	No	AC
Planar Biped Locomotion	Off-line	Real	No	AC
Robot T-ball	Off-line	Real	No	NAC
Motor Skill Coordination	On-line	Real	No	PoWER
Traffic Control	On-line	Simulated	Yes	NAC

Table 1: A summary of the practical applications described in this section.

This action decreases convergence times and is especially recommended for all those *risky* tasks where the robot needs to start interaction with a real environment from a *safe* position.

3 POLICY GRADIENT METHODS FOR ROBOT CONTROL

In this chapter, we analyze and propose the utilization of different PG techniques for AUV in real robotic tasks. When dealing with real robotics, most of the RL methodologies are implemented as reactive behaviors inside a control architecture. A brief discussion of the evolution of behavior-based control architectures [12] for real robots is given at the beginning of this chapter. It first reviews the history of control architectures for autonomous robots, starting with traditional methods of Artificial Intelligence (AI) and ending with the actual most used behavior-based architectures. The generalization problem, which highly affects real robotics, is described next. The most common approaches to confront this problem and their application to robotics tasks are overviewed. A simple PG algorithm is chosen to carry out the first experimental tests. The GPOMDP algorithm is one of the algorithms proposed in this thesis to build a policy for an RLP in a real autonomous underwater task. To solve the generalization problem, the policy is first approximated by means of an ANN and secondly by a barycentric interpolator. A more complex algorithm is proposed for the second set of results: the NAC. The great performance demonstrated by the NAC algorithm indicates it as the learning algorithm used for the final approach of this thesis. Also, in order to speed up the learning process various techniques are briefly discussed and, among them, a two-step learning process which shares simulated and real learning is chosen as the best option to reduce convergence time. The final proposal, which represents the main contribution of this thesis, is the application in a real underwater robotic task of a two step RL technique such as NAC. For this purpose, the NAC algorithm is first trained in a simulated environment where it can quickly build an initial policy. In the second step the policy is transferred to the real robot to continue the learning process on-line in a real environment. All the experimental results are shown in Chapter 6.

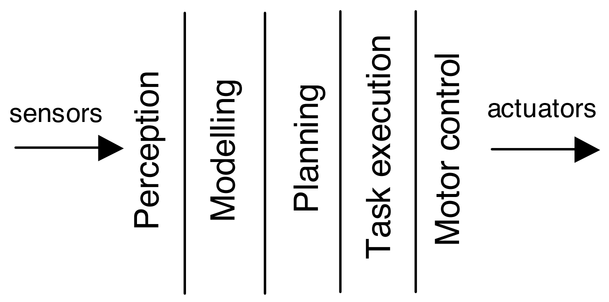

Figure 6: Phases of a classical deliberative control architecture [34].

3.1 EVOLUTION OF CONTROL ARCHITECTURES

The first attempt at building autonomous robots began around the mid-twentieth century with the emergence of Artificial Intelligence. The approach begun at that time was known as *Traditional AI*, *Classical AI* or *Deliberative approach*. Traditional AI relied on a centralized world model for verifying sensory information and generating actions in the world, following the Sense, Plan and Act (SPA) pattern [106]. Its main architectural features were that sensing flowed into a world model, which was then used by the planner, and that plan was executed without directly using the sensors that created the model. The design of the classical control architecture was based on a top-down philosophy. The robot control architecture was broken down into an orderly sequence of functional components and the user formulated explicit tasks and goals for the system [31]. The sequence of phases usually found in a traditional deliberative control architecture can be seen in Figure 6.

Real robot architectures and programming using an SPA deliberative control architecture began in the late 1960s with the Shakey robot at Stanford University [45, 107]. As sensors, the robot was equipped with a camera, a range finder and bump sensors that translated the camera image into an internal world model. A planner took the internal world model and a goal and generated a plan that would achieve this goal. The executor took the plan and sent the actions to the robot. The robot inhabited a set of especially prepared rooms. It navigated from room to room, trying to satisfy a given goal. Many other robotic systems have been built with the traditional AI approach [6, 61, 80, 38, 78], all of which shared the same kind of problems. Planning algo-

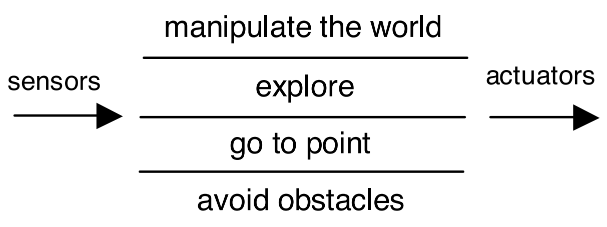

Figure 7: Structure of a behavior-based control architecture [34].

rithms failed with non-trivial solutions and the integration of the world representations was extremely difficult and, as a result, planning in a real world domain took a long time. Also, the execution of a plan without sensing was dangerous in a dynamic world. Only structured and highly predictable environments were proven to be suitable for classical approaches.

In the middle of the 1980s, due to dissatisfaction with the performance of robots in dealing with the real world, a number of scientists began rethinking the general problem of organizing intelligence. Among the most important opponents to the AI approach were Rodney Brooks [30], Rosenschein and Kaelbling [130] and Agre and Chapman [3]. They criticized the symbolic world which Traditional AI used and wanted a more reactive approach with a strong relation between the perceived world and the actions. They implemented these ideas using a network of simple computational elements, which connected sensors to actuators in a distributed manner. There were no central models of the world represented explicitly. The model of the world was the real one as perceived by the sensors at each moment. Leading the new paradigm, Brooks proposed the *Subsumption Architecture*. A subsumption architecture is built from layers of interacting finite-state machines. These finite-state machines were called *Behaviors*, representing the first approach to a new field called *Behavior-based Robotics*. The behavior-based approach used a set of simple parallel behaviors which reacted to the perceived environment proposing the response the robot must make in order to accomplish the behavior (see Figure 7). Whereas SPA robots were slow and tedious, behavior-based systems were fast and reactive. There were no problems with world modeling or real-time processing because they constantly sensed the world and reacted to it.

Since then, behavior-based robotic approaches have been explored in depth. The field attracted researchers from many disciplines such as biologists, neuroscientists, philosophers, linguists, psychologists and, of course, people working with computer science and artificial intelligence, all of whom found practical uses for this approach in their various fields of endeavor. Successful robot applications were built using the behavior-based approach, most of them at MIT [39, 60]. A well known example of behavior-based Robotics is Arkin's motor-control schemas [11], where motor and perceptual schemas were dynamically connected to one another.

Despite the success of the behavior-based models, they soon reached their limits in terms of capabilities. Limitations when trying to undertake long-range missions and the difficulty of optimizing the robot behavior were the most important difficulties encountered. Also, since multiple behaviors can be active at any given time, behavior-based architectures need an *arbitration* mechanism that enables higher-level behaviors to override signals from lower-level behaviors. Therefore, another difficulty has to be solved: how to select the proper behaviors for robustness and efficiency in accomplishing goals? In essence, robots needed to combine the planning capabilities of the classical architectures with the reactivity of the behavior-based architectures, attempting a compromise between bottom-up and top-down methodologies. This evolution was named *Layered Architectures* or *Hybrid Architectures*. As a result, most of today's architectures for robotics follow a hybrid pattern. Usually, a hybrid control architecture is structured in three layers: the reactive layer, the execution control layer and the deliberative layer (see Figure 8). The reactive layer takes care of the real time issues related to the interactions with the environment. It is built by the robot behaviors where sensors and actuators are directly connected. Each behavior can be designed using different techniques, ranging from optimal control to reinforcement learning techniques. The execution control layer interacts between the upper and lower layers, supervising the accomplishment of the tasks. This layer acts as an interface between the numerical reactive and the symbolic planning layers. It is responsible for translating high-level plans into low-level behaviors and for enabling/disabling the behaviors at the appropriate moment with the correct parameters. Also, the execution

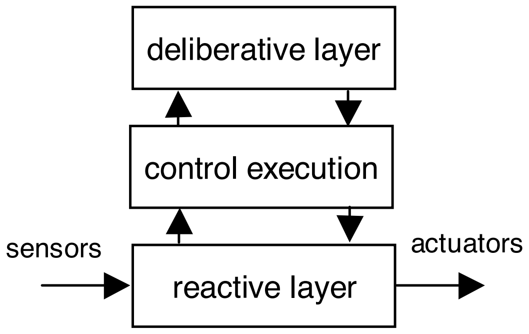

Figure 8: The hybrid control architecture structure [34].

control layer monitors the behaviors being executed and handles any exceptions that may occur. The deliberative layer transforms the mission into a set of tasks which make up a plan. It determines the long-range tasks of the robot based on high-level goals.

One of the first architectures which combined reactivity and deliberation was proposed by James Firby. In his thesis [46], the first integrated three-layer architecture is presented. From there, hybrid architectures have been widely used. One of the best known is Arkin's Autonomous Robot Architecture (AURA) [13], where a navigation planner and a plan sequencer were added to the initial behavior-based motor-control schemas architecture. The Planner-Reactor architecture [83] and the Atlantis [50] used in the Sojourner Mars explorer are well known examples of hybrid architectures.

Nowadays, hybrid architectures represent the basic foundation for control architectures on real robotic systems, but what about behavior programming? This section has described the behaviors as a set of *primitives* belonging to the reactive layer directly responsible for interaction with the environment. Therefore, in terms of AI, there is a key factor that must be taken into serious consideration when designing behaviors: *adaptation*. Intelligence cannot be realized without adaptation. If a robot requires autonomy and robustness it must adapt itself to the environment. The need for adaptation is obvious when dealing with real robotics. The programmer does not know all the parameters of the system and the robot must be able to perform in different and changing environments. At the moment there is no established methodology to develop adaptive behavior-based systems. The next section describes the applicability of RL techniques to design behaviors for control architectures.

3.2 REINFORCEMENT LEARNING BASED BEHAVIORS

RL is a very suitable technique for learning in unknown environments. As has been described in Chapter 2, RL learns from interaction with the environment and according to a scalar reward value. The reward function evaluates the environment state and the last taken action with respect to achieving a particular goal. The mission of an RL algorithm is to find an optimal state/action mapping which maximizes the sum of future rewards whatever the initial state is. The learning of this optimal mapping or policy is also known as the RLP.

The features of RL make this learning theory useful for robotics. There are parts of a robot control system which cannot be implemented without experiments. For example, when implementing a reactive robot behavior, the main strategies can be designed without any real test. However, for the final tuning of the behavior, there will always be parameters which have to be set with real experiments. A dynamics model of the robot and environment could avoid this phase, but it is usually difficult to achieve this model with reliability. RL offers the possibility of learning the behavior in real-time and avoid the tuning of behaviors with experiments. RL automatically interacts with the environment and finds the best mapping for the proposed task, which in this example would be the robot's behavior. The only necessary information which has to be set is an initial parameterization of the policy, the reinforcement function which gives the rewards according to the current state and the past action. It can be said that by using RL the robot designer reduces the effort required to implement the whole behavior, to the effort of designing the reinforcement function. This is a great improvement since the reinforcement function is much simpler and does not contain any dynamics. There is another advantage in that an RL algorithm can be continuously learning and, therefore, the state/action mapping will always correspond to the current environment. This is an important feature in changing environments.

RL theory is usually based on FMDPs. However, in a robotic system, it is usual to measure signals with noise or delays. If these signals are related to the state of the environment, the learning process will be damaged. In these cases, it would be better to consider the environment as a POFMDP (see Section 2.2). The dynamics of the environment is formulated as a POFMDP and the RL algorithms use the properties of these systems to find a solution to the RLP. TD techniques are able to solve the RLP incrementally and without knowing the transition probabilities between the states of the FMDP. In a robotics context, this means that the dynamics existing between the robot and the environment do not have to be known. As far as incremental learning is concerned, TD techniques are able to learn each time a new state is achieved. This property allows the learning to be performed online, which in a real system context like a robot, can be translated to a real-time execution of the learning process. The term *online* is here understood as the property of learning with the data that is currently extracted from the environment and not with historical data.

The combination of RL with a behavior-based system has already been used in many approaches. In some cases, the RL algorithm was used to adapt the coordination system [84, 48, 67, 90]. Moreover, some researchers have used RL to learn the internal structure of the behaviors [132, 158, 154, 140] by mapping the perceived states to control actions. The work presented by Mahadevan [85] demonstrates that breaking down a robot control policy into a set of behaviors simplifies and increases the learning speed. In this dissertation, PG techniques are designed to learn the internal mapping of a reactive behavior.

The main problem of RL when applied to a real system is the *generalization* problem. In a real system, the variables (states or actions) are usually continuous. However, RL theory is based on FMDPs, which uses discrete variables. Classic RL algorithms must be modified to allow continuous states or actions. The next section overviews common methodologies applied to solve the generalization problem from its theoretical point of view.

3.3 GENERALIZATION METHODS

As previously stated, the generalization problem appears when dealing with real environments with continuous states and/or actions. In order to successfully adapt continuous variables to finite TD methods, the first solution might be to discretize the continuous space into a finite space. However, in order to solve real robotic tasks, accurate discretizations would require a high number of states or state/action pairs which makes such discretizations impractical for solving real tasks. The need for function approximators that closely *match* or approximate a target function in a task-specific way seems obvious.

The reasons why there are several techniques for solving the generalization problem is because none offer a perfect solution. Some techniques have a higher generalization while others are computationally faster, but their most important feature is their capability to converge into an optimal policy. Convergence proofs have only been demonstrated for algorithms using tabular representations [149, 42, 160, 143]. In addition, in order to maintain stability, the learning procedure must be performed on-policy. This means that off-policy methods using linear function approximators cannot profit from such convergence proof and may diverge [121, 17, 159]. Also, TD methods suffer from convergence problems if the function approximator is not selected properly, as they estimate the VF based on immediate rewards and on the function itself (boostrapping) which means that the VF will always be an approximation of the discrete function. Several methods have been developed to deal with divergence problems [17, 51, 159] either using special update rules or special function approximators to ensure convergence. However, their use is restricted in practice since their convergence times are slow and they have limited generalization capabilities. Despite the lack of convergence proofs, there are many successful algorithms including linear and non-linear approximations with both on and off-policy methodologies [81, 173, 151, 40]. Although it is not intended to be a survey, the next lines show common function approximation techniques, some of which have been used in this thesis to deal with the generalization problem.

Classical approaches for function approximation in RL are based on lookup table methods. However, lookup tables do not scale well with the number of inputs, and huge continuous spaces of robotic tasks make them impractical for the real world. Working in this direction, a very intuitive approach to solving the generalization is offered by *Decision Trees* [105]. This methodology discretizes the entire state or the state/action space, also called *root*, into a space with a different resolution distribution. Those regions of the space requiring more accuracy would be divided into more cells or *leaves* than others requiring less. This variable resolution substantially reduces the number of finite states or state/action pairs. Decision trees highly improve the generalization problem with respect to classical tabular RL methods which use a uniform discretization. In addition, the convergence of the algorithms is sometimes proved and the generalization capability has been shown to be very high [102]. However, in order to select greedy actions, the whole set must be tested, resulting in slow convergence times. Moreover, decision trees always use a finite set of actions and, therefore, the generalization is only carried out in the state space. Several proposals using decision trees to solve the generalization problem can be found. The *G-Learning* algorithm [37] applies a decision tree to represent the QL over a discrete space. In another approach, the *Continuous U-tree* applies ideas similar to the G-Learning to a simulated robot environment, but with a continuous representation of the state. Other proposals show their results in simulated tasks [123, 102].

A popular technique for solving the generalization problem in RL is the Cerebellar Model Articulation Controller (CMAC) [4, 5]. CMAC is a simple linear approximator based on a set of *tiles*. The input space is divided up into hyper-rectangles, each of which is associated with a memory cell or tile. The contents of the memory cells are the parameters of *weights*, which are adjusted during training. Usually, more than one quantization of input space is used. Various layers or *tilings* may be superimposed so that any point in input space is associated with a number of tiles, and therefore with a number of memory cells. The output of a CMAC is the algebraic sum of the weights in all the memory cells activated by the input point. The generalization capability of CMAC depends on the number of tilings and the number of tiles within each tiling. The higher the number of tiles the better the resolution and the higher the number of tilings the better the generalization. Since CMAC is a linear approximator, convergence is guaranteed as long as it used with an on-policy algorithm. However, successful applications of both on and off-policy techniques can be found [151, 134, 133, 163]. Drawbacks of CMAC are similar to those presented by decision trees. Although a high generalization capability has been demonstrated by this technique, its application to real robotics shows slow convergence.

Other function estimation methodologies are *Memory-based* techniques, also known as *Instance-based* techniques [15]. Memory-based methods comprise a family of learning algorithms that, instead of performing explicit generalization, compare new problem instances with instances seen in training and stored in memory. Each element of the memory represents a visited state or state/action pair also called a *case*. Each case contains the value of the approximated function. The memory is initialized with zero elements and is dynamically filled according to the visited cases. If a new case which is not contained in the memory appears, neighbor cases are used to compute the value of the new case. There are different techniques to estimate the value of new cases: *nearest neighbors*, *weighted average* and *locally weighted regression* are some of them. The convergence of memory-based methods is not guaranteed, although its generalization capability is very high. Memory-based systems are a kind of *lazy learning* since the generalization beyond the training data is delayed until a query is made to the system. The main advantage gained in employing a lazy learning method is that the target function will be locally approximated and, therefore, lazy learning systems can simultaneously solve multiple problems and deal successfully with changing environments. The main disadvantage of these techniques is that they have high computational requirements. Each time a new approximation is required, the whole process must be performed with all its neighbor cases computation. Also, the space required to store the entire training data set tends to be high. Particularly noisy training data increases the case base unnecessarily because no abstraction is made during the training phase. In addition, as happened with decision trees, in order to find a greedy action, a finite set of actions must be evaluated, which again implies extra computation time. Some examples of their performance on simulated tasks can be found in [113, 93, 109]. Despite their inconvenience for real robotics, some approaches obtained good results [144, 97]. A comparison between memory-based methods and CMAC can be found in [134] where memory-based techniques show better performance.

Continuing along the path of linear approximators, a similar CMAC technique is the *basis function* [152]. The idea is to substitute the binary activation function of each tile by a continuous function. The approximated value is a linear function of the parameter vector to be identified. Corresponding to each state, there is a vector of *features* with the same number of components as the parameter vector. The resultant output of the function is the algebraic sum of the products between each parameter with its correspondent feature. Feature selection becomes critical for a good approximation and the later success of the learning algorithm. Choosing features appropriate to the task is an important way of adding prior domain knowledge to reinforcement learning systems. Intuitively, the features should correspond to the natural features of the task, those for which generalization is most appropriate. If we are valuing geometric objects, for example, we might want to have features for each possible shape, color, size, or function. If we are valuing states of a mobile robot, then we might want to have features for locations, degrees of remaining battery power, recent sonar readings, and so on. The features may be constructed from the states in many different ways, resulting in different kinds of well known basis function approximators: *coarse coding*, *tile coding* [152], *radial basis functions* [120, 118], *barycentric interpolators* [101, 102] and more recently *Proto-value Functions* [86] are some examples. The main advantage of using basis functions is that, if feature selection is appropriate for the task, they can be very efficient in terms of both data and computation for real robotic applications. Over the last few years, much work has been devoted to the topic of developing basis functions techniques. The methods presented in [145] and [122] are essentially heuristics that attempt to exploit the intuition that trajectories taken by the system may lie along a low-dimensional manifold. In [95] the basis functions and the parameters vector are adapted simultaneously, using either gradient descent or the cross-entropy method. In this case, though, the resulting approximator can no longer be considered linear. Also, some practical applications of radial basis functions in robotics can be found in [135, 75].

One of the most important breakthroughs was the introduction of Artificial Neural Networks (ANNs) as function approximators for learning algorithms [58]. An ANN is a parameterized function composed of a set of *neurons* which become activated depending on some input values. These neurons generate an output according to an *activation* *function*. Neuron outputs can be used as inputs for other neurons, forming a multi-layered structure. By combining a set of neurons and using non-linear activation functions, an ANN is able to approximate any non-linear function, making them a very powerful algorithm. As a drawback, the convergence of an RL algorithm using an ANN cannot be guaranteed due to its non-linear nature. In addition, ANNs suffer from *inference* problems [167, 15]. When an ANN-based learning algorithm updates its parameters to change the output value, that change affects the entire space, so a learning step in a particular region of the state space causes a step backward in another place. Solutions to avoid inference point towards using ANNs locally or uniformly updating the space. ANN-based algorithms have been successfully applied to approximate VFs in real robotic tasks [53, 56, 87, 49, 32] and higher generalization capabilities are demonstrated in [173, 27]. In [156] an ANN based system learns to play backgammon. The resultant algorithm was able to play at the same level as the best human players in the world. Successful applications like this motivated the application of function approximators to real robotics. Initial experiments presented in this dissertation use ANN as the function approximator to learn the policy.

Over the last few years, another class of algorithms have received special attention from the RL community: Support Vector Machines (SVMs) [162]. These algorithms have been used mainly for classification tasks. Given a set of training examples, each marked as belonging to one of two categories, an SVM training algorithm builds a model that predicts whether a new example falls into one category or the other. Intuitively, an SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall on. Although these methods are known for their application in classification, they can also be used to approximate functions. The main idea is to apply a variation of SVM called SVM-Regression or Support Vector Regression (SVR) [139] to approximate the VF. SVR represents a powerful alternative: in principle unharmed by the dimensionality, trained by solving a well defined optimization problem, and with generalization capabilities presumably superior to local instance-based methods. Even though most of the experimental results are limited to off-line learning [43], some on-line algorithms have appeared recently with promising results [89, 65].

3.4 GPOMDP IN ROBOTICS

The GPOMDP algorithm is an on policy search methodology. Its aim is to obtain a parameterized policy that converges to an optimal by computing gradient approximations of the averaged reward from a single path of a controlled POFMDP. The theoretical aspects of the algorithm have been presented in Section 2.6.2. The GPOMDP algorithm is a TD method and, like all TD methods, the dynamics of the environment does not have to be known. Another important feature of TD algorithms is that the learning process can be performed on-line. The characteristics of the algorithm together with its mathematical simplicity make the GPOMDP approach a good startup to solve the RLP in real robotic tasks. This section analyzes the application of the on-line version of the algorithm in a real system such as a robot.

The algorithm works as follows: having initialized $T > 0$, the parameter vector θ_0 arbitrarily and setting the eligibility trace vector $z_0(\theta) = 0$, the learning procedure will be iterated T times. At every iteration, the system receives the current state from the environment s_t. The algorithm generates a control action a_t according to current stochastic policy $\pi(a_t|s_t, \theta_t)$ and a reward r_{t+1}. The reward function $r(s, a)$ is defined experimentally depending on the RLP. After that, the eligibility trace $z_t(\theta)$ is updated according to the gradient of the current policy with respect to its parameters. The eligibility's decay factor λ is set between $[0, +1)$ with the aim of increasing or decreasing the agent's memory of past actions. The immediate reward received r_{t+1} and the eligibility trace $z_{t+1}(\theta)$ allow us to finally compute the new vector of policy parameters θ_{t+1}. The current policy is directly modified by the new parameters, becoming a new policy to be followed by the next iteration, getting closer to a final policy that represents a correct solution to the problem. The learning rate factor α, tuned experimentally, controls the step size of the iteration process, trying to find a compromise between learning speed and convergence guarantees.

From the different generalization techniques described in Section 3.3, two different function approximators have

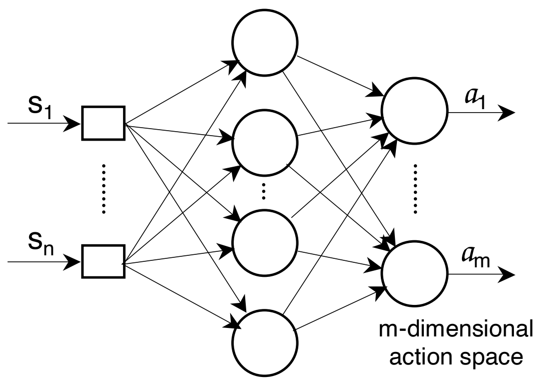

Figure 9: Implementation of a simple policy with an ANN.

been selected to represent the policy: ANNs and barycentric interpolators.

3.4.1 *Policy Approximation with ANN*

Together with the GPOMDP algorithm, a parameterized ANN function is one of the approximators selected to represent a stochastic policy $\pi(a_t|s_t, w_t)$. Its weights w_t represent the policy parameters to be updated at every iteration step. Advantages and drawbacks of various function approximators have been described in Section 3.3. The main reason for choosing an ANN as one of the approximators for these initial experiments is for its excellent ability to approximate any nonlinear function in comparison with other function approximators. Also, an ANN is easy to compute and the required number of parameters is very small, making it suitable for application with the GPOMDP approach. An ANN is a function approximator able to approximate a mathematical function which has a set of inputs and outputs. The input and output variables are real numbers and the approximated functions can be non-linear, according to the features of the ANN. ANN were inspired by the real neurons found in the human brain, although a simpler representation is used. As stated in Section 3.3, the basic theory of ANN was widely studied during the 1980s but there are still many active research topics.

The ANN model used to represent the policy in this first approach is depicted in Figure 9. The state vector is driven directly as the ANN input. The output of the network is the number of continuous variables which make up the action space. As can be seen, neurons are grouped in different *layers*. The first layer uses the state vector as neuron inputs $\{s_1, ..., s_n\}$. This set of inputs is also called *input layer*, although it is not a layer of neurons. The second and consecutive layers use as neuron inputs the neuron outputs from the preceding layer. Finally, the last layer of the ANN is the *output layer*, where each neuron generates an output of the network as actions $\{a_1, ..., a_m\}$ of the approximated policy. All the neuron layers preceding the output layer are also called *hidden layers* since the neuron output values are not seen from the outside nor are they significant.

Going deeper into the design of the network, Figure 10 takes a closer look into a single neuron. The neuron j located in the layer l has a set of inputs $\{y_1^{l-1}, y_2^{l-1}, ..., y_p^{l-1}\}$ and one output y_j^l. The value of this output depends on these inputs, on a set of weights $\{w_{j1}^l, w_{j2}^l, ..., w_{jp}^l\}$ and on an *activation function* φ^l. In the first computation, the induced *local field* v_j^l of the neuron j is calculated by adding the products of each input y_i^{l-1} by its corresponding weight w_{ij}^l. An extra input y_0^{l-1} is added to the computation of v_j^l. This input is called the *bias* term and has a constant value equal to 1. By adjusting the weight w_{j0}^l, the neuron j can be activated even if all the inputs are equal to 0. The local field v_j^l is then used to compute the output of the neuron $y_j^l = \varphi^l v_j^l$. The activation function has a very important role in learning efficiency and capability. The learning process of the ANN consists of adapting the parameters or weights of the network until the output is equal to a desired response. The GPOMDP algorithm has the goal of indicating the procedure to modify the values of these parameters.

The next lines will describe the parameter update procedure carried out by the GPOMDP algorithm for the particular case of an ANN as the policy approximator. Once the parameters of the ANN are initialized, the network receives an observation of the state as input s_t and gives a control action as output a_t. Then, the algorithm is driven to another state s_{t+1} and will receive a reward associated with this new state r_{t+1}. The first step in the parameter update procedure is to compute the gradient of the policy with respect to each parameter. As defined in Equation 3.1, in the ANN context, the gradient of the policy with respect to each parameter is

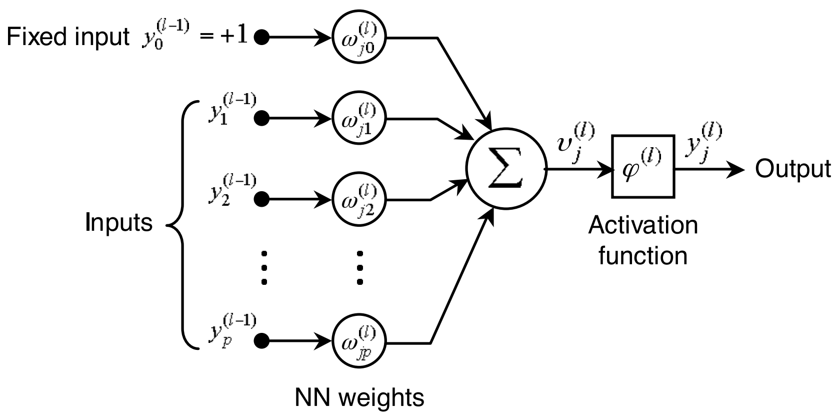

Figure 10: Diagram of a single neuron j located at layer l.

equal to the *local gradient* δ_t associated with a single neuron multiplied by the input to the neuron y_t.

$$\frac{d\pi(a_t|s_t, w_t)}{dw_t} = \delta_t y_t. \tag{3.1}$$

In order to calculate the local gradient of all the neurons in a particular ANN, the local gradient of the neurons at the output layer must be computed first and then propagated to the rest of the neurons in the hidden layers by means of *backpropagation*. For the local neuron j located in the output layer, we may express its local gradient as

$$\delta_j^o = e_j \varphi'(o_j). \tag{3.2}$$

Here e_j is the error in the output of neuron j, $\varphi'(o_j)$ corresponds to the derivative of the activation function associated with that neuron and o_j is the function signal in the output for that neuron. Therefore, for neuron j located in a hidden layer, the local gradient is defined as follows

$$\delta_j^h = \varphi'(o_j) \sum_k \delta_k w_{kj}. \tag{3.3}$$

As can be seen in Figure 11, when computing the gradient of a neuron located in a hidden layer, the previously obtained gradient of the following layers must be back propagated. $\varphi'(o_j)$ is the derivative of the activation function associated with that neuron and the summation term includes the different gradients of the following neurons

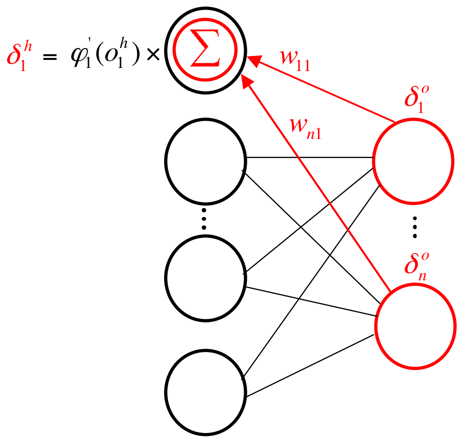

Figure 11: Local gradient computation for a hidden layer neuron.

back propagated by multiplying each gradient δ_k with its correspondent weight w_{kj}.

With the local gradients of all the neurons computed, the gradients of the policy with respect to each weight in the ANN can be obtained. As described in the GPOMDP procedures of Algorithm 4, eligibility traces can be calculated following Equation 3.4. Finally, the parameter vector is updated following Equation 3.5, i. e.,

$$z_{t+1}(w) = \lambda z_t(w) + \delta_t y_t, \tag{3.4}$$

$$w_{t+1} = w_t + \alpha r_{t+1} z_{t+1}(w). \tag{3.5}$$

Here the vector of parameters w represents the network weights to be updated and r_{t+1} is the reward given to the agent at every time step. The learning rate α controls the step size of the iteration process.

3.4.2 *Policy Approximation with Barycentric Interpolators*

A parameterized basis function represented by a barycentric interpolator is another approximator chosen to represent the policy $\pi(a_t|s_t, \theta_t)$ together with the GPOMDP approach. Barycentric interpolators, described in Section 3.3, are a specific class of basis function approximators. These basis functions are a popular representation for VFs because they provide a natural mechanism for variable resolution discretization of the function and the barycentric co-ordinates

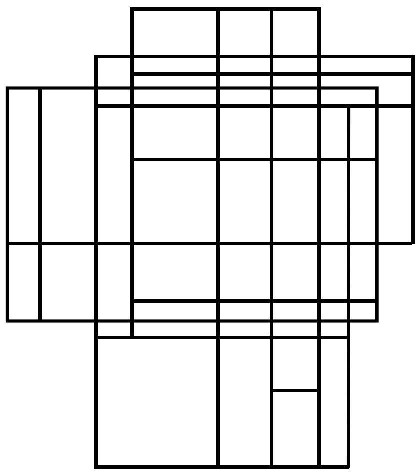

Figure 12: An N-dimension rectangular mesh. The mesh elements need not be regular, however, the boundary can vary in extent.

allow the interpolators to be used directly by value iteration algorithms. Another advantage in choosing linear function approximators is their convergence guarantees when used together with an on-policy method such as the GPOMDP algorithm. Barycentric interpolators apply an interpolation process based on a finite set of discrete points that form a mesh. This mesh does not need to be regular, but the method outlined here assumes that the state space is divided into a set of rectangular boxes. An example of an appropriate mesh is shown in Figure 12. In practice, we will often start with a grid that is iteratively remeshed and which would look more like Figure 13. Getting the barycentric interpolation itself to work only requires a mesh of the type shown in Figure 12, though. The key is that any particular point is enclosed in a rectangular box that can be defined by the 2^N nodes in our N-dimensional state space.

As shown in Figure 14, ξ_i being a set of nodes distributed in a rectangular mesh for any state s. Once it is enclosed in a particular box $(\xi_1, ..., \xi_4)$ of the mesh, the state s becomes the *barycenter* inside this box with positive coefficients $p(s|\xi_i)$ of sum 1 called the *barycentric coordinates* and, therefore,

$$s = \sum_{i=1..4} p(s|\xi_i)\xi_i. \tag{3.6}$$

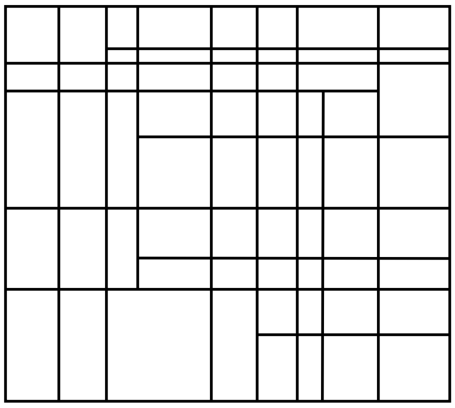

Figure 13: A more typical-looking mesh, where an initial grid has been refined through iterative splitting of some of the elements.

Thus, $V(\xi_i)$ is set as the value of the function at the nodes previously defined as ξ_i. Having defined the value of the function at the nodes of the mesh, Equation 3.7 shows how to compute the value of the function at state s, called the *barycentric interpolator* of state s and defined as $V(s)$. The value of state s is calculated as a function of the value $V(\xi_i)$ at the nodes of the box in which the state is enclosed and its barycentric coordinates $p(s|\xi_i)$. As depicted in Figure 15, we give

$$V(s) = \sum_{i=1..4} p(s|\xi_i)V(\xi_i). \tag{3.7}$$

The policy is approximated using a barycentric interpolator function where the nodes of the mesh $V(\xi_i)$ represent the policy parameters θ to be updated at each learning iteration. Therefore, given an input state s_t, the policy will compute a control action $V(s_t) = a_t$. The parameters of the approximator are updated following the procedures stated in Algorithm 4. As defined in Equation 3.8, in the barycentric approximator context, the gradient of the policy with respect to each parameter is equal to the derivative of the approximator with respect to the parameter multiplied by an error function e_t, hence,

$$\frac{d\pi(a_t|s_t,\theta_t)}{d\theta_t} = \frac{dV(s_t)}{dV(\xi_i)}e_t. \tag{3.8}$$

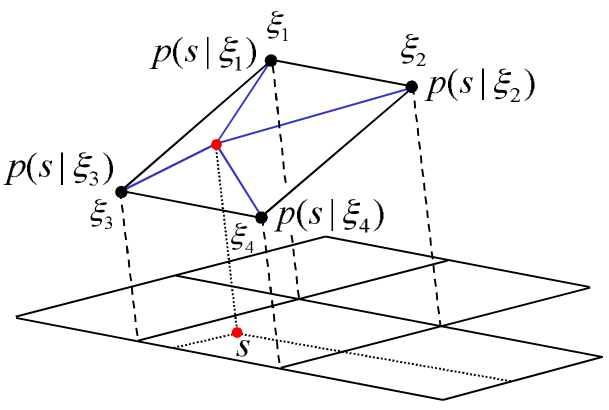

Figure 14: Graphic representation of the *barycentric coordinates* given a state s in a 2 dimensional mesh case.

Here the error is given by

$$e_t = V(s_t)_{\text{desired}} - V(s_t). \tag{3.9}$$

The eligibility trace z_{t+1} is updated following Equation 3.10. Eligibility's decay factor λ controls the memory on past actions. The gradient of the policy with respect to a particular parameter contains the barycentric coordinate of the parameter $p(s|\xi_i)$,

$$z_{t+1} = \lambda z_t + p(s_t|\xi_i)e_t. \tag{3.10}$$

Finally, the new parameter vector is given by

$$V(\xi_i)_{t+1} = V(\xi_i)_t + \alpha r_{t+1} z_{t+1}. \tag{3.11}$$

The vector $V(\xi_i)$ represents the policy parameters to be updated, r_{t+1} is the reward given to the agent at every time step and α as the learning rate of the algorithm.

3.5 NATURAL ACTOR-CRITIC IN ROBOTICS

The experimental results obtained with both approximations described in Section 3.4 showed poor performance of the algorithm. Although the GPOMDP approach is able to converge to an optimal or near optimal policy, results suggest that the application of a *pure* PG algorithm in a real task is quite slow and rigid. Fast convergence and adaptation to an unknown changing environment is a must when

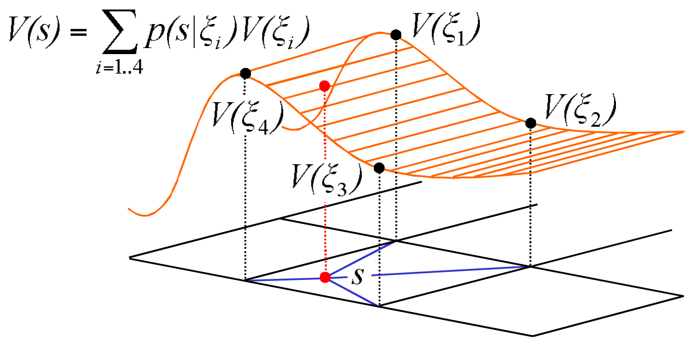

Figure 15: Calculation of the function approximator output given a particular state s.

dealing with real applications and a PG seems not able to do so alone. With the objective of finding a faster algorithm, the attention of this thesis moves to AC methodologies. As discussed in Section 2.8, AC methods try to combine the advantages of PG algorithms with VF methods. The actor's PG gives convergence guarantees while the critic's VF reduces variance of the policy update improving the convergence rate of the algorithm. Also, AC methods are on-policy algorithms and, therefore, the learning procedure benefits from convergence proofs. The main features of AC methods seem to have what real robotics is looking for.

The theoretical aspects of the algorithm have been presented in Section 2.8.4. The algorithm is divided into two main blocks, one concerning the critic and another the actor. The critic is represented by a VF $V^\pi(s)$ which is approximated by a linear function parameterization represented as a combination of the parameter vector v and a particularly designed basis function $\phi(s)$, the whole expression being $V^\pi(s) = \phi(s)v$. On the other hand, the actor's policy is specified by a normal distribution $\pi(a|s) = \mathcal{N}(a|Ks, \sigma^2)$. The mean μ of the actor's policy distribution is defined as a parameterized linear function of the form $a = Ks$. The variance of the distribution is described as $\sigma = 0.1 + 1/(1 + \exp(\eta))$. Therefore, the whole set of the actor's parameters is represented by the vector $\theta = [K, \eta]$. The actor's policy derivative has the form $\frac{d \log \pi(a|s,\theta)}{d\theta}$ and, as detailed in Section 2.6.4, in order to obtain the natural gradient, this function is inserted into a parameterized compatible function approximation, $f(s,a)^\pi_w = \frac{d \log \pi(a|s,\theta)}{d\theta} w$, with the parameter vector w as the true gradient .

At every iteration, action a_t is drawn from the current policy π_t generating a new state s_{t+1} and reward r_t. After updating the basis function and the critic's statistics by means of LSTD-Q(λ), the VF parameters ν and the natural gradient w are obtained. The actor's policy parameters are updated only if the angle between two consecutive natural gradients is small enough compared to an ϵ term. The learning rate of the update is controlled by the α parameter. Next, the critic has to forget part of its accumulated statistics using a forgetting factor, $\beta \in [0, 1]$. The current policy is directly modified by the new parameters becoming the new policy to be followed in the next iteration, getting closer to a final policy that represents a correct solution to the problem.

3.6 METHODS TO SPEEDING UP REINFORCEMENT LEARNING

Speeding up RL algorithms in real continuous high dimensional domains represents a key factor. The idea of providing initial high-level information to the agent is a topic that has received significant attention in the real robotic field over the last decade. The main objective of these techniques is to make learning faster in contrast to tedious reinforcement learning methods or trials-and-error learning. Also, it is expected that the methods, being user-friendly, would enhance the application of robots in human daily environments. This section briefly describes some common techniques for enhancing and accelerating learning in real robotic applications.

One form of speeding up the learning process is to have the robot learn a particular task by *watching* the task being performed by a human or an expert system. The approach is known as *Learning From Demonstration*, *Programming From Demonstration*, *Imitation Learning* or simply *Teaching* [28, 15, 81]. Teaching plays a critical role in human learning. The main idea is that teaching can shorten the learning process and even turn intractable learning tasks into tractable. If learning is resumed as a search problem, the teacher gives external guidance for this search. For this kind of learning, the robot can try to repeat the actions taken by the teacher or learn how the teacher acts and reacts to the different situations encountered, finally building its own policy. Although these kinds of techniques encompass a wide range of algorithms with thier own procedures, a general teaching process overview may be one like this: The teacher shows the learning agent how an instance of the target task can be achieved from some initial state. The sequence of actions, state transitions and rewards are recorded as a *lesson*. Several taught lessons can be collected and repeatedly replayed. Like experienced lessons, taught lessons should be replayed selectively; in other words, only policy actions are replayed. But if the taught actions are known to be optimal, all of them can be replayed all the time. The term lesson refers to both taught and experienced lessons. It is unnecessary for the teacher to demonstrate only optimal solutions in order for the agent to learn an optimal policy. In fact, the agent can learn from both positive and negative examples. This is an important property as it makes teaching techniques different from *supervised learning* approaches [100, 119]. Teaching is useful for two reasons: First, teaching can direct the agent to first explore the promising part of the search space which contains the goal states. This is important when the search space is large and a thorough search is infeasible. Second, teaching can help the agent to avoid being stuck in local maxima. The idea of providing initial high-level information to the agent has achieved great success in several applications. In [57], an outdoor mobile robot learns to avoid collisions by observing a human driver operate a vehicle equipped with sensors that continuously produce a map of the local environment. The work presented in [117] shows how imitation learning can be used to extract an initial policy to learn a control task where an anthropomorphic arm tries to hit a ball. First, a human teacher gives the agent an insight into the task by helping it hit the ball. Results show that this initial policy is not good enough to fulfil the task. Therefore, in a second phase, an NAC algorithm is applied to improve the policy obtaining a better policy which successfully accomplishes the task.

Other approaches to accelerate the learning process with real robots use *Supplied Control Policies* to feed prior knowledge to the agent [144]. The main idea is to split the learning into two phases. In the first phase, the robot is being controlled by a supplied control policy. This initial control policy can be represented either by a classic controller or by a human controlling the robot. During this period, the agent passively watches state transitions, actions and rewards that the supplied control policy generates and uses them to update its policy. The supplied control policy exposes the agent to those areas of the space-state where the reward is not zero, accelerating the learning process. Differing from previously described techniques, the agent does not mimic the trajectories generated by the supplied policy, it just uses them to update its policy. Also, these trajectories are not generated directly, as in teaching, but as a result of interaction with the environment. When the learned policy is considered good enough, in a second phase, the agent takes control of the robot and continues the learning by normal interaction with the environment. Results presented in [147] show the viability of supplied control policies for speeding up learning algorithms. In that paper, a mobile robot successfully learns control policies for two simple tasks: obstacle avoidance and corridor following.

Another way of reusing past experiences, which was investigated in the DYNA architecture [150], is to use experiences to build an *action model* and use it for planning and learning. An action model can be understood as a computer simulator which mimics the behaviors of the environment for a particular RLP. The action model contains the dynamics of the environment together with its reward function. Instead of direct interaction with the environment, the agent can experience the consequences of actions without physically being in the real world. As a result, the robot will learn faster as the iteration steps with the simulator would be faster than those with the real world and more importantly, fewer mistakes will be made when the robot starts running in the real world with a previously trained policy. Approximate policies learned in simulation represent a good startup for the real learning process, meaning that the agent will face the challenge of the real world with some initial knowledge of the environment, avoiding initial gradient plateaus and local maxima dead ends [81]. The help of computer simulators has been successfully applied in [16] where a PG algorithm designed to control a RC-helicopter is first trained off-line by means of a computer simulator. Once the agent's policy is considered to be *safe* enough it is transferred to the real vehicle where the learning process continues. Also, in [77], a technique called *Time Hopping* is proposed for speeding up reinforcement learning algorithms in simulation. Experiments on a simulated biped crawling robot confirm that this technique can greatly accelerate the learning process.

The success of such techniques is closely related to the accuracy of the simulation, being almost impossible to apply to real complex tasks where a model of the environment can not be obtained. In order to have a simulation phase that really represents an advantage, a good model has to be provided. For the particular RLP proposed in this thesis, most of the model identification work is based on the dynamic equations of motion derived from the Newtonian or Lagrangian mechanics [47], which are characterized by a set of unknown parameters. The application of system identification techniques to underwater vehicles is concerned with the estimation, on the basis of experimental measurements, of a number of parameters or of hydrodynamic derivatives that characterize the vehicle's dynamics [2]. Such measurements, collected during full-scale trials by the on-board sensors [33], or during captive testing in a planar motion mechanism [108], are processed by a parameter estimation routine [82]. Several methods have been proposed in the literature for this purpose like the use of genetic algorithms and simulated annealing [157]. Nevertheless, since the dynamic equation of motion can be formulated as an equation linear in the vector of unknown parameters, LS methods can be easily used.

The final proposal of this thesis is the application in a real underwater robotic task of RL-based behavior in a two step learning process. The RL method chosen is the NAC algorithm. For this purpose, the learning algorithm is first trained in a simulated environment where it can quickly build an initial policy. An approximated model of the environment emulates a robot with the same number of DoFs of the real one. Once the simulated results are accurate enough and the algorithm has acquired enough knowledge from the simulation to build an approximated policy, in a second step the policy is transferred to the real robot to continue the learning process on-line in a real environment. Since the simulated model and environment are just approximations, the algorithm will have to adapt to the real world. An important idea is that, once the algorithm converges in the real environment, the learning system will continue working forever, being able to adapt to any future change. The next chapter accurately describes the identification procedures carried out to obtain a model of the environment, which is applied to learn an initial policy during the first step of the learning process.