From Single-Agent to Multi-Agent Reinforcement Learning: Foundational Concepts and Methods

Learning Theory Course

Gonçalo Neto

Instituto de Sistemas e Robótica
Instituto Superior Técnico

May 2005

Abstract

Interest in robotic and software agents has increased a lot in the last decades. They allow us to do tasks that we would hardly accomplish otherwise. Particularly, multi-agent systems motivate distributed solutions that can be cheaper and more efficient than centralized single-agent ones.

In this context, reinforcement learning provides a way for agents to compute optimal ways of performing the required tasks, with just a small instruction indicating if the task was or was not accomplished.

Learning in multi-agent systems, however, poses the problem of non-stationarity due to interactions with other agents. In fact, the RL methods for the single agent domain assume stationarity of the environment and cannot be applied directly.

This work is divided in two main parts. In the first one, the reinforcement learning framework for single-agent domains is analyzed and some classical solutions presented, based on Markov decision processes.

In the second part, the multi-agent domain is analyzed, borrowing tools from game theory, namely stochastic games, and the most significant work on learning optimal decisions for this type of systems is presented.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The use of artificial agents, wether software or robotic, to solve problems and accomplish tasks is becoming a commonplace, as technologies evolve and new challenges are presented. Robots, for example, are already around for a few decades and have become essential in may branches of industry, particularly the automotive one.

The problem of programming an agent to do a certain task is sometimes hard, but programming it to decide over groups of possible tasks stems a different problem, one that is, nevertheless, more and more essential as there is a constant need to provide the agents with some autonomy. An example are robots in situations where man cannot access easily or has little communications: search and rescue scenarios like the World Trade Center, planetary missions like the Mars rovers Spirit and Opportunity, etc. The problem of imbuing agents with a decision-making mechanism can, sometimes, be hard to tackle as the designer himself does not have a clear idea of what the agent should do, or does not have the time to provide all the possible acting options. Hence, the need for learning techniques arises.

Agents can be characterized by the following components: perception, reasoning and actuation. In this context, *reinforcement learning* is a learning framework that is very tied to the concept of agent. The motivation is that, as stated before, the programmer usually does not have the clear notion of what the agent's actions should be and is only able to decide wether a complete task was or was not well accomplished. So the agent cannot be told the sequence of actions to do but rather wether a sequence of actions was or was not good enough.

Reinforcement learning, instead of testing each possible sequence to find the appropriate one, which probably would be a very inefficient procedure,

acts by using the concept of state and iteratively propagating the reward it receives at the end of the task to the actions on the chosen sequence. It is, in this sense, a set of methods to learn decision making policies while performing the task and interacting with the environment.

The concept of *Markov decision process* will be essential in modeling the learning task and providing a framework over which reinforcement learning methods can be constructed.

A harder problem than the one of an agent learning what to do is when several agents are learning what to do, while interacting with each other. Research on this problem is an interesting one as the fields of multi-agent learning and multi-agent robotics are increasingly proving to be a necessity for many applications. Tackling the problem with one robot is interesting and necessary but, typically, several agents can accomplish tasks that a single one would not, or would do so in a costly manner.

Learning is also essential in this domain but poses additional problems to the learning methods. In single agent cases the environment was considered to be stationary but that is not possible to do in the multi-agent scenario because the other agents themselves are changing the environment. The problem is the one of learning in a non-stationary environment, possibly with the other agents also learning.

To solve such problems, some attention has been given to *game theory* and, particularly, the framework of *stochastic games* and the decision problems arising from it.

The objective of this work is to provide a survey of the basic frameworks, results and algorithms in both single and multi-agent learning. For this, it was divided in two main chapters, plus a conclusion:

- Chapter 2 looks into the single agent learning problem. It starts by presenting Markov decision processes (MDPs) and the optimality concepts associated with the framework. Afterwards, it looks into *dynamic programming* algorithms used to solve and understand MDPs. Finally, it presents some classical reinforcement learning algorithms and an extension which relates them to Monte Carlo approaches.

- Chapter 3 describes the learning problem and associated concepts in the multi-agent domain. It tries to explain some ideas from game theory, such as matrix games and stochastic games, giving some insight on classical methods for solving the first. The problem of solving stochastic games is essentially the most addressed problem in the multi-agent

learning community. So, this work then presents two classes of algorithms that act as cornerstones in the multi-agent learning research.

- Chapter 4 tries to draw some comparative conclusions on the methods analyzed and describes further topics of reinforcement learning, for single agent systems, which have been widely addressed although there are still some open problems, and multi-agent systems, which remain mostly as research directions and challenges for researchers in the area.

Chapter 2

Single-Agent Framework

The single agent reinforcement learning framework is based on the model of Figure 2.1, where an agent interacts with the environment by selecting actions to take and then perceiving the effects of those actions, a new state and a reward signal indicating if it has reached some goal (or has been penalized, if the reward is negative). The objective of the agent is to maximize

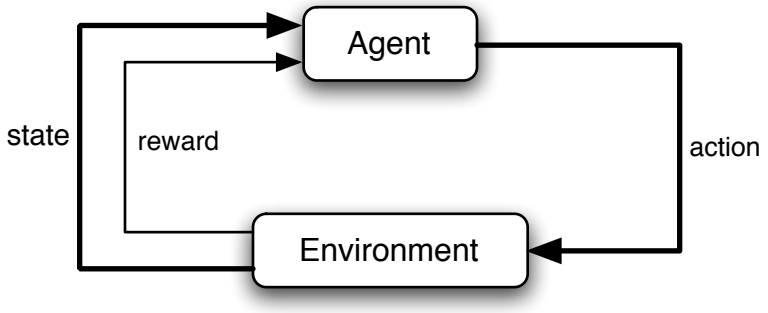


Figure 2.1: The general reinforcement learning task.

some measure over the rewards, like the sum of all rewards after a number of actions taken.

This general idea can be described by the framework of *Markov decision processes*, over which the solutions for the reinforcement learning problem are constructed.

2.1 Markov Decision Processes

Markov Decision Processes (Bellman, 1957; Howard, 1960; Bertsekas, 1995; Sutton and Barto, 1998; Puterman, 1994) are, in fact, the foundation for much of the research on agent control. They can be defined as a tuple (S, A, T, R) where:

- A is an action set.
- S is a state space.
- $T : S \times A \times S \longrightarrow [0, 1]$ is a transition function defined as a probability distribution over the states. Hence, we have $T(s, a, s') = Pr\{s_{t+1} = s' | s_t = s, a_t = a\}$. s_{t+1} represents the state of the process at time $t+1$, s_t the state at time t and a_t the action taken after observing state s_t.
- $R : S \times A \times S \longrightarrow \mathcal{R}$ is a reward function representing the expected value of the next reward, given the current state s and action a and the next state a': $R(s, a, s') = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\}$. In this context r_{t+1} represents the immediate payoff of the environment to the agent at time $t + 1$.

The fact that there are no time dependences either on T or on R is due to the stationarity (by the definition) of the MDP. Although this is not stated in the definition, reinforcement learning algorithms without a generalization mechanism are usually designed to work on *finite MDPs*, that is, MDPs in which both the state and the action spaces are finite. So, unless clearly stated otherwise, the term MDP will refer to a finite Markov decision process.

The idea of MDPs is that the agent acts on the environment with some action a, in state s, and waits for the response of the environment, in the form of the following state s' and a real number representing the immediate reward the agent receives by choosing to perform a in s.

The task of deciding which action to choose in each state is done by a *policy* function. Generally, a policy is a collection of probability distributions, one for each trace of the the process – $\pi(s_t, a_{t-1}, s_{t-1}, a_{t-2}, \dots) \in PD(A)$ – defining the probability that each action will be chosen for that particular trace of the system. However, there is no need to consider other than Markovian policies because the MDP itself is Markovian by construction – it is sufficient to define the policy for each state of the MDP.

A policy can also be thought as a projection transforming the MDP in an induced discrete-time Markov chain. The interesting thing with this idea

is that the Markov chains theoretic paraphernalia becomes available and can act as a performance measure for the current policy. In (Bhulai, 2002) the two frameworks are related and the results applied to fields traditionally related to Markov chains such as control of queuing systems.

Optimality Concepts

The goal of an agent living in an environment that can be modeled as a Markov Decision Process is to maximize the expected reward over time, which on itself aggregates a myriad of formulations. The most common criteria are:

Finite-horizon model: in this scenario the agent tries to maximize the sum of rewards for the following M steps:

$$E\left\{\sum_{k=0}^{M} r_{t+k+1}\middle| s_t = s\right\}$$

The objective is to find the best action, considering there are only M more steps in which to collect rewards.

Infinite-horizon discounted reward model: in this scenario the goal of the agent is to maximize reward at the long-run but favoring short-term actions:

$$E\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}\middle| s_t = s\right\} \quad , \quad \gamma \in [0,1[$$

The discount factor γ regulates the degree of interest of the agent: a γ close to 1 gives similar importance to short-term actions and long-term ones; a γ close to 0 favors short-term actions.

Average reward model: in this model the idea is to find actions that maximize average reward on the long-run:

$$\lim_{M\to\infty} E\left\{\frac{1}{M}\sum_{k=0}^{M} r_{t+k+1}\middle| s_t = s\right\}$$

This model makes no distinction between policies which take reward in the initial phases from others that shoot for the long-run rewards.

The first criterion could be used to model systems where there's a hard deadline and the task has to be finished in M steps. In reinforcement learning, usually the adopted model is the *Infinite-horizon discounted reward model*, probably not only because of its characteristics but because it bounds the sum. The single agent (MDP) algorithms surveyed here almost all adopt this model, although there is some work on finding policies that maximize other models.

A fundamental concept of algorithms for solving MDPs is the *state value function*, which is nothing more than the expected reward (in some reward model) for some state, given the agent is following some policy:

$$V^{\pi}(s) = E\left\{\sum_{k=0}^{\infty}\gamma^{k}r_{t+k+1}\middle|s_t = s, \pi\right\} \tag{2.1}$$

Similarly, the expected reward given the agent takes action a in state s and following policy π could also be defined:

$$Q^{\pi}(s,a) = E\left\{\sum_{k=0}^{\infty}\gamma^{k}r_{t+k+1}\middle|s_t = s, a_t = a, \pi\right\} \tag{2.2}$$

This function is usually know as *Q-function* and the corresponding values as *Q-values*.

From Equation 2.1 a recursive relation can be derived, which will act as the base of much of the ideas behind dynamic programming and reinforcement learning algorithms to solve MDPs.

$$\begin{aligned}
V^{\pi}(s) &= E\left\{\sum_{k=0}^{\infty}\gamma^{k}r_{t+k+1}\middle|s_t = s, \pi\right\} \\
&= E\left\{r_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^{k}r_{t+k+2}\middle|s_t = s, \pi\right\} \\
&= \sum_{a}\pi(s,a)\sum_{s'}T(s,a,s')\left[R(s,a,s') + \gamma E\left\{\sum_{k=0}^{\infty}\gamma^{k}r_{t+k+2}\middle|s_{t+1} = s', \pi\right\}\right] \\
&= \sum_{a}\pi(s,a)\sum_{s'}T(s,a,s')\left[R(s,a,s') + \gamma V^{\pi}(s')\right]
\end{aligned} \tag{2.3}$$

Note that the term $\pi(s,a)$ is used to represent the probability of taking action a in state s. In some situations, the term $\pi(s)$ will be used to represent

the action selected by policy π. Generally, this a probabilistic outcome, except for deterministic policies where π can be thought as $\pi : S \rightarrow A$.

The resulting recursive equation, called the *Bellman equation*, has a unique solution for each policy which is the state value function for that policy (Howard, 1960).

From the Bellman equation a relation between state values and Q-values can be derived resulting in:

$$Q^{\pi}(s,a) = \sum_{s' \in S} T(s,a,s') \left[R(s,a,s') + \gamma V^{\pi}(s') \right] \tag{2.4}$$

and

$$V^{\pi}(s) = \sum_{a} \pi(s,a) Q^{\pi}(s,a) \tag{2.5}$$

As previously stated, the goal of solving a MDP is usually to find a policy that guarantees a maximal reward, with some given reward criterion. Using state values, a policy π' is said to dominate a policy π if and only if, for every s in the state space, $V^{\pi'}(s) > V^{\pi}(s)$. An optimal policy is one which is *undominated* in the sense that no other can expect to do better, in any state. An optimal policy π^* is always guaranteed to exist[1] and sometimes even more than one, although they share the same value function, which can be defined as:

$$V^*(s) = \max_{\pi} V^{\pi}(s) \ , \ \forall_{s \in S} \tag{2.6}$$

or, for Q-values:

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) \ , \ \forall_{s \in S} \forall_{a \in A} \tag{2.7}$$

Each optimal policy must respect the Bellman consistency relation expressed in Equation 2.3 as well as equations derived from it, particularly, the ones expressed in Equations 2.5 and 2.4. Applying Equation 2.5 to the optimal policy, the optimal equation can be written as a maximization over the action space rather than over the policy space. Notice that for all π in the policy space $\sum_a \pi(s,a) Q^*(s,a) \leq \max_a Q^*(s,a)$.

$$V^*(s) = \max_{a} Q^*(s,a) \tag{2.8}$$

[1] This can be proved by construction, choosing at each state the policy that maximizes V for that state and discarding the rest of the policy. On the other hand, a maximum value of V always exists in the infinite-horizon discounted reward model due to $\gamma < 1$.

Combining it with Equation 2.4 it is possible to obtain the *Bellman optimality equations*, for state values and for Q-values respectively:

$$V^*(s) = \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s') \right] \tag{2.9}$$

$$Q^*(s,a) = \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q(s',a')^* \right] \tag{2.10}$$

This last equation makes the task of finding optimal policies really easy, when the optimal Q-values are know. In fact, the consequence of the Bellman optimality equations is that an optimal policy is one that will always choose an action that maximizes the Q-function for the current state. Several policies exist, when different actions have the same Q-value (the Q-function is unique for a given MDP) for a given state, but all of them are *greedy* with respect to the Q-function. Another consequence it that there is always a deterministic policy which is optimal, in the infinite-horizon discounted reward model.

The problem is that, usually, there's no knowledge of the optimal state function or Q-function and, so, there's a need for methods which estimate such functions and find optimal policies from them. Dynamic programming methods are used to accomplish this task when the parameters of the MDP are know, particularly the functions R and T. When the agent does not have such knowledge and has to find optimal policies based solely on experience,[2] Monte Carlo or reinforcement learning algorithms are appropriate.

2.2 Dynamic Programming

Solving MDPs have always been closely tied to the idea of *dynamic programming*, which was introduced by (Bellman, 1957) as a possible solution to a wide variety of problems. From then on, a lot of research has been done in that area and extensive treatments of the subject can be found in texts like (Bertsekas, 1995; Bertsekas and Tsitsiklis, 1996; Ross, 1983). An interesting approach is the one of (Cassandras and Lafortune, 1999) which relates dynamic programming with other methods for controlling *discrete event systems*. It has been also widely used in *optimal control* applications.

Two classical dynamic programming methods for MDPs are *value iteration* and *policy iteration*.

[2] In the MDP sense: in each state, choosing an action, observing the next state and collecting the reward.

2.2.1 Value Iteration

As stated previously, a way of finding an optimal policy is to compute the optimal value function. *Value iteration* is an algorithm to determine such function, which can be proved to converge to the optimal values of V. The core of the algorithm is:

$$V(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V(s') \right] \tag{2.11}$$

Note that the expression was obtained by turning the Bellman optimality equation (2.9) into an update rule. An important result about value iteration is that it is guaranteed to find an optimal greedy policy in a finite number of steps, even though the optimal value function may not have converged – usually the optimal policy is found long before the value function has converged. In (Bertsekas, 1987) those questions are discussed, along with some convergence proofs.

One thing lacking in the definition of the algorithm is the termination rule – it is not obvious when the algorithm should stop. A typical stopping condition bounds the performance as a function of the *Bellman residual* of the current value-function. That is the approach taken by (Sutton and Barto, 1998) in their description of the algorithm and it states that if:

$$\max_{s \in S} |V_{k+1}(s) - V_k(s)| < \epsilon$$

then

$$\forall_{s \in S} \quad |V_{k+1}(s) - V^*(s)| < \frac{2\epsilon\gamma}{1-\gamma}$$

Another stopping criterion, which may lead the algorithm to stop some iterations earlier, is discussed in (Puterman, 1994).

Assuming the Bellman residual criterion is used, the complete algorithm can be seen in Algorithm 2.1.

Although the algorithm assumes a full sweep through the state space before passing to the next iteration, the assignments of V do not need to be done through successive sweeps. *Asynchronous dynamic programming* algorithms, originally proposed by (Bertsekas, 1982; Bertsekas, 1983) who also called them *distributed dynamic programming* algorithms, back up the values of the states in an indefinite order and, in fact, the value of a state can be backed up several times before the value of another one gets backed up even once. The condition for convergence is that all states are backed up infinitely often. These algorithms were further discussed in (Bertsekas and Tsitsiklis, 1989).

Algorithm 2.1 Value iteration

Initialize $V(s)$ arbitrarily

repeat
  $\delta \leftarrow 0$
  for all $s \in S$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s')\right]$
    $\delta \leftarrow \max(\delta, |v - V(s)|)$
  end for
until $\delta < \epsilon$

2.2.2 Policy Iteration

Another way of finding an optimal policy in a finite MDP is by manipulating the policy directly rather than finding it through the state values. A simple algorithm for doing that is based on the idea of alternating two different steps: a *policy evaluation* step plus a *policy improvement* step.

In the first one, the state values corresponding to the starting policy are computed based on an iterative expression which, similarly to value iteration, is taken from the Bellman equation, although this time not for the optimal values:

$$V^{\pi}(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^{\pi}(s')\right] \tag{2.12}$$

The discussion of when to stop the algorithm is the same as for value iteration. A different option is to use the Bellman equation directly and solve a set of linear equations to find the value function corresponding to the policy being used. This could not be done so easily for value iteration because the equations are not linear (due to the max operator).

In the improvement step, the current policy is analyzed to see if it is greedy with respect to the current value function. It can be proved that by making the policy greedy it will be strictly improved over the current policy, except when the policy is already optimal. This also provides a termination criterion for the policy improvement algorithm[3].

The algorithm, explicitly solving the policy evaluation equations at each step of the algorithm, is presented in Algorithm 2.2. In this situation the policy is deterministic and, so, $\pi(s)$ represents an action.

[3]The termination criterion is for the overall algorithm, which still does not solve the

Algorithm 2.2 Policy iteration

Initialize $V(s)$ and $\pi(s)$ arbitrarly

repeat

Policy Evaluation
Solve the system of linear equations
$V^{\pi}(s) = \sum_a \pi(s,a) \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^{\pi}(s')\right]$

Policy Improvement
$optimalPolicy? \leftarrow TRUE$
for all $s \in S$ **do**
$b \leftarrow \pi(s)$
$\pi(s) \leftarrow \arg\max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V(s')\right]$
if $b \neq \pi(s)$ **then**
$optimalPolicy? \leftarrow FALSE$
end if
end for

until $optimalPolicy? = TRUE$

The policy evaluation step could, of course, be substituted by an iterative procedure similar to value iteration but based on Equation 2.12, solving the linear equations numerically.

Considering that the number of deterministic policies over a finite MDP is also finite (in fact the number of policies is equal to $|A|^{|S|}$), and knowing the policy improvement step always strictly improves the policy, there is an bound in the total number of iterations. Policy Iteration could be claimed to be better than value iteration because it usually takes fewer iterations but, on the other hand, value iteration does not consume so much time in each iteration – policy iteration has to solve a possibly large set of equations or use an iterative procedure similar to value iteration itself, and do it several times. In the end, it is not clear which of the algorithms is better.

2.2.3 Generalized Policy Iteration

An idea explored in (Sutton and Barto, 1998) is that of *generalized policy iteration* (GPI), in which much of the algorithms of dynamic programming and even reinforcement learning can de fitted. The term GPI refers to a process of evaluation interacting with a process of improvement, with the

problem for the policy evaluation step.

evaluation process being used to estimate a value function with respect to some policy and the improvement process being used to make a better policy, given some state values. Figure 2.2 illustrates such a process.

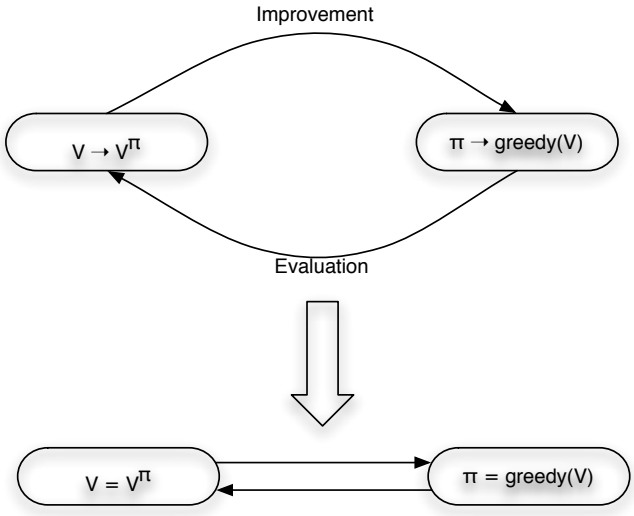


Figure 2.2: The idea of generalized policy iteration.

This notion is easily seen in policy evaluation but value iteration also fits in such a description. In fact, after each run through the state space (the evaluation step) the policy inherently improves to a new greedy policy over the current state value function, by choosing the greedy action in the next run – the improvement step is then implemented.

2.3 Learning with Model-free methods

The dynamic programming methods presented in the previous section are intended to learn optimal value functions (and from them find optimal policies) in the presence of a model of the system. In fact, both value iteration and policy iteration used explicitly the T and R functions which, in general, could be unknown, particularly if the objective of the method is to learn the optimal policy while interacting with the environment.

In reinforcement learning, generally it is assumed the model is unknown and, in this situation, two approaches can be pursued. A *model-based* approach tries to learn the model explicitly and then use methods like dynamic

programming to compute the optimal policy with respect to the estimate of the model. On the other hand, a *model-free* approach concentrates on learning the state value function (or the Q-value function) directly and obtaining the optimal policy from this estimates.

The focus of this work will be only on model-free methods for learning in MDPs. The class of algorithms used to accomplish such a task is generally known as *temporal difference methods*, as defined by (Sutton, 1988).

2.3.1 TD(0): estimating the value function

Temporal difference methods share a common characteristic: similarly to value iteration or policy evaluation, they use the estimates for the state values of other states to update the estimate of the current state, which is generally know as *bootstrapping*. The simplest temporal difference method *TD(0)* was introduced by (Sutton, 1988) and has the following update rule:

$$V(s) \leftarrow V(s) + \alpha \left(r + \gamma V(s') - V(s) \right) \tag{2.13}$$

The state s' is observed and the reward r collected after applying policy π to state s. This method works as the policy evaluation step in the sense that it estimates the value function for a given policy but does not improve the policy.

The idea behind this procedure is that the quantity $r + V(s')$ acts as an estimate for $V(s)$ and, probably, it is closer to the state value, for policy π, because it is computed based on an immediate reward, incorporating knowledge from the environment into the estimate. In fact, the term weighed by α is nothing more than an estimate of the error between the current state value function and the real state value function, for the given policy. This procedure can be viewed as some sort of *stochastic approximation*, although most of the proofs of convergence for temporal difference learning methods were done without relating it to the theory behind stochastic approximation. The first move to bring both bodies of knowledge together was made by (Tsitsiklis, 1994).

This type of procedure is also said to make a *sample backup* because the new estimate of the value function is obtained based on a sample of the possible options for the next state. The dynamic programming algorithms, on the other hand, are said to perform a *full backup*, relying on the estimates of all the possible successor states.

For the method to converge, it has to visit every state infinitely often and the learning rate α has to be slowly decreased, which was proved in (Sutton, 1988). The algorithm version of the method is presented in Algorithm 2.3.

Algorithm 2.3 TD(0) learning

Initialize $V(s)$ arbitrarly
Choose the policy π to be evaluated

Initialize s
loop
  $a \leftarrow$ probabilistic outcome of pdf $\pi(s)$
  Take action a, observe reward r and next state s'
  $V(s) \leftarrow V(s) + \alpha\left(r + \gamma V(s') - V(s)\right)$
  $s \leftarrow s'$
end loop

In this algorithm, the task is considered to continue indefinitely, that is, having no terminal states – this situation is know as *continuing tasks*. If, on the other hand, the MDP has terminal states and the task naturally blocks in some state after some iterations, which is called *episodic tasks*, the algorithm has to be restarted in some initial state but preserving the already computed estimates of the state value function. A possible option is to initialize $V(s)$ not arbitrarily but with the current estimate of $V(s)$, for all $s \in S$.

There is an important characteristic which divides the temporal difference learning methods into two main classes: *on-policy* methods and *off-policy* methods. In the first class, the policy used for control of the MDP is the same which is being improved and evaluated. In the later, the policy used for control, called the *behavior policy*, can have no correlation with the policy being evaluated and improved, the *estimation policy*. The dilemma of on-policy vs off-policy methods is related with the dilemma of exploration vs exploitation, which will be further addressed in Section 2.4.

2.3.2 Q-learning: off-policy control

The *Q-learning* method, proposed by (Watkins, 1989), is perhaps the most popular and widely used form of reinforcement learning, mainly due to the ease of implementation. It is an *off-policy* method which learns optimal Q-values, rather than state-values, and simultaneously determines an optimal policy for the MDP. The use of Q-values is justified due to the need of selecting an action based on them: similarly to $TD(0)$, in each iteration there is only knowledge of two states, s and one of its successors, and it is not possible to make a full backup as it was done in dynamic programming

algorithms. So, Q-values provide some insight on the future quality of the actions in the successor state and make the task of choosing an action easier.

The update rule for Q-learning is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\big) \tag{2.14}$$

which is based on the same idea as Equation 2.13 in the sense that $r + \gamma \max_{a'} Q(s',a')$ is a sample for $Q(s,a)$

In (Watkins and Dayan, 1992) the method is proved to converge to the Q-values for the optimal policy, Q^*, if two convergence conditions were met:

1. Every state-action pair has to be visited infinitely often.
2. α must decay over time such that $\sum_{t=0}^{\infty} \alpha_t = \infty$ and $\sum_{t=0}^{\infty} {\alpha_t}^2 < \infty$.

The algorithm form can be seen in Algorithm 2.4. Note that the update rule is always performed based on a greedy and deterministic policy which is being improved. On the other hand, the actions chosen for control are based on some other policy. In fact, the behavior policy could have no relation with $Q(s,a)$, for example, a policy with a uniform distribution over the action space could be used to generate the actions. However, the objective is to control the MDP while learning the optimal policy and by always using the random behavior policy usually a poor control is achieved, although the estimation policy converges to the optimal one anyway.

Algorithm 2.4 Q-learning

Initialize $Q(s,a)$ arbitrarly

Initialize s
loop
  $a \leftarrow$ probabilistic outcome of *behavior* policy derived from $Q(s,a)$
  Take action a, observe reward r and next state s'
  $Q(s,a) \leftarrow Q(s,a) + \alpha\big(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\big)$
  $s \leftarrow s'$
end loop

$\forall_{s \in S} \quad \pi(s) \leftarrow \arg\max_a Q(s,a)$

2.3.3 Sarsa: on-policy control

Sarsa was first introduced by (Rummery and Niranjan, 1994) as an on-policy method for learning the optimal policy while controlling the MDP. In

this situation, the algorithm behaves according to the same policy which is being improved and, so, the update rule is as follows:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big(r + \gamma Q(s',a') - Q(s,a)\big) \tag{2.15}$$

The method relies on information about the variables s, a, r, s', a' and that explains the origin of the name Sarsa, introduced by (Sutton, 1996).

Convergence results can be seen in (Singh *et al.*, 2000) and, basically, require that each state-action pair is visited infinitely often and that the policy being used converges to a greedy policy. Examples of policies which can be used to meet the requirement can be seen in Section 2.4.

The algorithm form is presented in Algorithm 2.5. Note that the policy used to choose the actions is the same as the one used for evaluation and improvement. As stated before, in the Q-learning algorithm the policy used for evaluation and improvement was greedy, which is identified by the use of the max operator in computing the new estimate for $Q(s,a)$.

Algorithm 2.5 Sarsa

Initialize $Q(s,a)$ arbitrarly

Initialize s
$a \leftarrow$ probabilistic outcome of policy derived from Q
loop
  Take action a, observe reward r and next state s'
  $a' \leftarrow$ probabilistic outcome of policy derived from Q
  $Q(s,a) \leftarrow Q(s,a) + \alpha\big(r + \gamma Q(s',a') - Q(s,a)\big)$
  $s \leftarrow s'$ and $a \leftarrow a'$
end loop

2.4 Exploration vs Exploitation

The methods presented previously, when used for learning and control, all share a common characteristic: for the algorithm to converge to the optimal policy, all the possible alternatives have to visited infinitely often. This condition allows the convergence to avoid being stuck on sub-optimal policies.

With an off-policy method this is always achieved in the limit by using *soft* behavior policies, meaning that no action has a 0 probability of being chosen. On-policy methods, on the other hand, require that the policy itself

converges to a greedy one, while performing exploration. This is generally achieved by making a succession of soft policies converge to the greedy.

In both situations, the dilemma of the learning method is wether to focus on using the already acquired information for control or trying to get new information, which could lead to better policies and a better control. The bottom line is: reinforcement learning algorithms have to act while learning.

A simple option for balancing exploration and exploitation is to have the behavior policy be a mixture of an *exploration policy* and an *exploitation policy* as stated by:

$$\Pi_{behavior} = \Gamma \cdot \Pi_{explore} + (1 - \Gamma) \cdot \Pi_{exploit} \tag{2.16}$$

where $0 < \Gamma < 1$ is a parameter which can be adjusted to allow more exploration or more exploitation. For example, on-policy methods just have to make $\Pi_{exploit}$ a greedy policy and modify so the parameter so that $\Gamma \to 0$ and, in the end, the remaining policy is greedy and is optimal because the corresponding values would also have converged to the optimal ones.

For off-policy methods, the general balancing expression can also be used and gamma may or not be modified, according to the specific needs of the algorithm. This and other considerations regarding exploration are surveyed in (Thrun, 1992).

2.4.1 ϵ-greedy policies

This kind of policies fit in the general description of Equation 2.16 and are basically a mixture of a greedy policy (for exploitation) and a uniform policy (for exploration). The parameter ϵ can be made arbitrarily small and drive the convergence to a greedy (and optimal) policy. Equation 2.17 describes such a policy.

$$\pi(s, a) = \begin{cases} \dfrac{\epsilon}{|A|} & \text{if } a \text{ is a greedy action,} \\ 1 - \epsilon + \dfrac{\epsilon}{|A|} & \text{otherwise.} \end{cases} \tag{2.17}$$

2.4.2 Softmax policies

Although ϵ-greedy policies are widely used in reinforcement learning, they have the disadvantage of giving equal weights to the non-greedy actions. Actually, some of them could be performing incredibly better than others, although they are not greedy. A way to avoid this problem is to use an utility function over the actions to better distinguish between them. This

kind of methods, called *softmax*, still give the highest probability to the greedy action but do not treat all the others the same way.

In the reinforcement learning context, the Q-function seems like a natural utility to use. The most common softmax method used in reinforcement learning relies on a Boltzmann distribution and controls the focus on exploration through a temperature parameter $\tau > 0$, as defined in Equation 2.18.

$$\pi(s,a) = \frac{e^{Q(s,a)/\tau}}{\sum_{a' \in A} e^{Q(s,a')/\tau}} \tag{2.18}$$

Similarly to ϵ-greedy policies, when $\tau \to 0$ the policy becomes greedy making the method adequate for the use with on-policy algorithms.

2.5 Eligibility Traces

The temporal differences methods presented previous are usually called *one-step* methods due to the fact that they perform their updates based on a one step backup. Using TD(0) as an example, the sample return for computing the estimation of state value at time t is:

$$R_t^{(1)} = r_{t+1} + \gamma V_t(s_{t+1})$$

There is no need to stick to this *one-step backup* and a *n-step backup* could be used like:

$$R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{n-1} r_{t+n} V_t(s_{t+n})$$

Generally, the sample return for backup could even be an average of samples with different step-sizes, for example:

$$R_t^a v = \tfrac{1}{2} R_t^{(1)} + \tfrac{1}{2} R_t^{(n)}$$

(Watkins, 1989) discusses this questions and proposes a weighted average over all the possible returns from time t to ∞:

$$R_t^\lambda = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)} \tag{2.19}$$

Of course this idea can not be implemented because it is not causal, needing information from time t to ∞. This interpretation is called by (Sutton and Barto, 1998) the *forward view*. The alternative is to use a counter, named an *eligibility trace*, which condenses information about the time it has passed since the last visit to a state and the total number of visits. This is called the *backward view* and they were proved to be equivalent by (Sutton, 1988).

2.5.1 TD(λ)

Extending TD(0) with eligibility traces can be made by defining the traces as:

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1 & \text{if } s = s_t \end{cases} \tag{2.20}$$

where $0 \leq \lambda \leq 1$ is the parameter controlling the decay. With this mechanism, each time a state is visited its counter is incremented by 1, after which it decreases exponentially.

The idea of the method is that the update error for the rule in Equation 2.13 now gets affected not only by α but by the corresponding eligibility trace as well, by using the following expression:

$$V(s) \leftarrow V(s) + \alpha\delta e(s) \tag{2.21}$$

where

$$\delta = r + \gamma V(s') - V(s) \tag{2.22}$$

Contrary to what has been done so far, with eligibility traces all the states have to be updated in each experience (choosing action a in state s and observing reward r and state s') and the information about the immediate payoff is propagated back to the states with higher eligibility traces.

As could be expected, when $\lambda = 0$ the algorithm reverts back to TD(0), updating only state s_t at time $t+1$. As for setting $\lambda = 1$, it is equivalent to doing a complete run of the method (which only makes sense for an episodic task) and in the end estimating the state values by collecting all the rewards and computing the total return. This procedure becomes very similar to a Monte-Carlo estimation of the state value function for the given policy and the equivalence between TD(1) and Monte Carlo methods was proved by (Sutton, 1988).

Convergence of the method for a general λ was proved in the mean by (Dayan, 1992) and with probability 1 by (Dayan and Sejnowski, 1994) or (Tsitsiklis, 1994).

The algorithmic form of TD(λ) can be seen in Algorithm 2.6.

2.5.2 Sarsa(λ)

The idea of eligibility traces can also be applied to Sarsa by using traces for each state-action pair and not just for each state. In fact, the update

Algorithm 2.6 TD(λ) learning

Initialize $V(s)$ arbitrarly and $e(s) = 0$, for all $s \in S$
Choose the policy π to be evaluated

Initialize s
loop
  $a \leftarrow$ probabilistic outcome of pdf $\pi(s)$
  Take action a, observe reward r and next state s'
  $\delta \leftarrow r + \gamma V(s') - V(s)$
  $e(s) \leftarrow e(s) + 1$
  for $\sigma \in S$ **do**
    $V(\sigma) \leftarrow V(\sigma) + \alpha\delta e(\sigma)$
    $e(\sigma) \leftarrow \gamma\lambda e(\sigma)$
  end for
  $s \leftarrow s'$
end loop

equations for both methods are very similar, except that Sarsa acts over Q-values. The traces for Sarsa(λ) can be defined as:

$$e_t(s,a) = \begin{cases} \gamma\lambda e_{t-1}(s,a) & \text{if } s \neq s_t \text{ or } a \neq a_t \\ \gamma\lambda e_{t-1}(s,a) + 1 & \text{otherwise} \end{cases} \tag{2.23}$$

and the update rule described by the following expression:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s) \tag{2.24}$$

where

$$\delta = r + \gamma Q(s',a') - Q(s,a) \tag{2.25}$$

The method was first explored in (Rummery and Niranjan, 1994; Rummery, 1995), although the convergence of the method for a general λ is yet to be proved.

Algorithm 2.7 presents the algorithmic version of this method.

2.5.3 Q(λ)

Q-learning has also been combined with eligibility traces giving birth to two different methods, one proposed in (Watkins, 1989) and the other in (Peng, 1993; Peng and Williams, 1996). The problem with applying eligibility traces

Algorithm 2.7 Sarsa(λ) learning

Initialize $Q(s,a)$ arbitrarly and $e(s,a) = 0$, for all $s \in S$ and $a \in A$
Choose the policy π to be evaluated

Initialize s
$a \leftarrow$ probabilistic outcome of policy derived from Q
loop
  Take action a, observe reward r and next state s'
  $a' \leftarrow$ probabilistic outcome of policy derived from Q
  $\delta \leftarrow r + \gamma Q(s',a') - Q(s,a)$
  $e(s,a) \leftarrow e(s,a) + 1$
  for $\sigma \in S$ and $\theta \in A$ **do**
    $Q(\sigma,\theta) \leftarrow Q(\sigma,\theta) + \alpha\delta e(\sigma,\theta)$
    $e(\sigma,\theta) \leftarrow \gamma\lambda e(\sigma,\theta)$
  end for
  $s \leftarrow s'$ and $a \leftarrow a'$
end loop

to Q-learning is knowing to which state-action pairs to assign credit for some sample return. Notice that as the policy used to control is not the same being evaluated and improved, non-greedy actions would be used to update a policy that only follows greedy actions.

Watkins's Q(λ) solves this problem by using the eligibility traces when the greedy action is followed by the behavior policy and reseting all traces to 0 when an exploratory action is chosen. However, another problem is created because the possible advantage of using eligibility traces is lost when the traces are reset.

Peng's Q(λ) on the other hand, never resets the traces and always follows the action that maximizes the Q-function. The backups, however, are different for one-step (the current state) than for all other states. This method has shown to perform better than Watkins's Q(λ) converging faster to the optimal policy, although there are no convergence proofs and this conclusions are based on empirical studies.

Regarding the advantages of methods using eligibility traces over one-step methods, they propagate information faster through the table of values and, hence, converge usually much faster to the optimal policies/values than their one-step counterparts. The catch is that they need an enormous

amount of computation over the others. A simple option to decrease such intensive computation is to keep a record of the states in which the eligibility trace is higher than a certain δ, considering all the others to be 0. Another approximation is to store only a fixed number of states, the ones with higher traces.

In practice, eligibility traces have proved to speed up the learning task and to be an useful tool when doing reinforcement learning applications, especially if there are many delayed reinforcements.

Chapter 3

Multi Agent Framework

The multi-agent framework is based on the same idea of Figure 2.1 but, this time, there are several agents deciding on actions over the environment. The big difference resides in the fact that all each agent probably has some effect on the environment and, so, actions can have different outcomes[1] depending on what the other agents are doing.

This is precisely the difference that poses problems when applying reinforcement learning techniques to the multi-agent domain. Usually, those are designed to solve stationary environments and, from the point of view of each agent, the environment is no longer stationary.

To model such a domain, the focus has turned to game theory, which is designed to solve multi-agent situations and in which the solutions involve compromises and cooperation. Particularly, the model most commonly used is that of *stochastic games* or the subclass of *matrix games*.

3.1 Matrix Games

The concept of *matrix games* (MG) or *strategic games* (von Neumann and Morgenstern, 1947; Owen, 1995) is a simple framework to deal with single-shot games, that is, games where there are multiple players but just one state with an associated reward structure. Formally, they can be defined as a tuple $(n, A_{1...n}, R_{1...n})$ where:

- n is the number of agents.

[1] In the single-agent case they could have different outcomes if T was not deterministic. Nevertheless, from the point of view of the agent the distribution is always the same, which does not happen in the multi-agent domain

- A_i is the action set for player i ($A = A_1 \times \cdots \times A_n$ is the joint action set)
- $R_i : A \to \mathcal{R}$ is the reward function of player i

One important characteristic of MGs is the fact that each agent's reward function depends on the actions of all the players and not just its own actions. The term matrix game arises from the fact that each player's reward structure can be represented as an n-dimensional matrix. As with MDPs, the type of games being considered have finite state and action spaces.

Games like Rock-Paper-Scissors (Table 3.1) or the Prisoner's Dilemma (Table 3.2) are examples of *two-person matrix games* (where $n = 2$). One usual convention in two-person games is that the first player always specifies the row index and the second player (the opponent in adversarial situations) specifies the column index. Each entry of the matrix specifies both players' rewards in the form: (player 1 reward , player 2 reward).

	Rock	Paper	Scissors
Rock	(0 , 0)	(-1 , 1)	(1 , -1)
Paper	(1 , -1)	(0 , 0)	(-1 , 1)
Scissors	(-1 , 1)	(1 , -1)	(0 , 0)

Table 3.1: Rock-Paper-Scissors reward matrix

	Tell	Not Tell
Tell	(-2 , -2)	(0 , -4)
Not Tell	(-4 , 0)	(-1 , -1)

Table 3.2: Prisoner's Dilemma reward matrix

In this framework, the concept of *strategy* plays a similar role to that of policy in MDPs. A strategy $\sigma_i \in PD(A_i)$ defines the way agent i decides on a Matrix Game. A collection of n strategies, one for each of the agents, is called a *joint strategy* and it can be written $\sigma = \langle \sigma_i, \sigma_{-i} \rangle$, where the notation σ_{-i} is used to refer to a joint strategy for all players except for player i.

Usually the name *pure strategy* is used for deterministic strategies and stochastic ones are generally called *mixed strategies*. For every joint strategy, there is an associated reward for each of the players $R_i(\sigma)$ which can defined in terms of the rewards for individual actions:

$$R_i(\sigma) = \sum_{a \in A} R_i(a) \prod_{j=1}^{n} \sigma_i(a_i) \tag{3.1}$$

Optimality Concepts

An individual strategy is said to be a *best-response strategy* if, for a given σ_{-i} played by all other players, it achieves the highest possible reward. We write:

$$\sigma_i \in BR_i(\sigma_{-i}) \tag{3.2}$$

where $BR_i(\sigma_{-i})$ represents the set of all best-response strategies of player i to σ_{-i} and $BR_i : PD(A_i) \rightarrow 2^{PD(A_i)}$ is usually called the *best-response function*[2] for player i. $\sigma_i^* \in BR_i(\sigma_{-i})$ if and only if:

$$\forall \sigma_i \in PD(A_i) \quad R_i(\langle \sigma_i^*, \sigma_{-i} \rangle) \geq R_i(\langle \sigma_i, \sigma_{-i} \rangle) \tag{3.3}$$

A *Nash equilibrium* is a collection of strategies, one for each player, that are best response strategies, which means that none of the players can do better by changing strategy, if all others continue to follow the equilibrium.

$$\forall_{i=1\ldots n} \quad \sigma_i \in BR_i(\sigma_{-i}) \tag{3.4}$$

An important characteristic of Matrix Games is that all of them have at least one Nash equilibrium.

A usual way of classifying Matrix Games is the following:

- *Zero-sum games* are two-player games ($n = 2$) where the reward for one of the players is always symmetric to the reward of the other player. Actually, this type of games is equivalent to *constant-sum games*, where the sum of both player rewards is always constant for every joint action. An example is Rock-Paper-Scissors, as shown in Table 3.1.

- *Team games* have a general number of players but their reward is the same for every joint action. An example is the Matrix Game shown in Table 3.3 that models a shared resource channel where the agents are playing cooperatively and the optimal is for one of them to always give priority to the other.

- *General-sum games* are all types of matrix games. However, the term is mainly used when the game can not be classified as a zero-sum one. An example is the Prisoner's Dilemma, as shown in Table 3.2.

[2] PD(A_i) represents the set of all probability distributions over A_i and $2^{PD(A_i)}$ its power set, that is, the set of all possible subsets of $PD(A_i)$

	Wait	Go
Wait	(0 , 0)	(1 , 1)
Go	(4 , 4)	(-2 , -2)

Table 3.3: A Team Game: Modeling of a Shared Resource

The first kind of games, also called *two-person zero-sum* games, is very appealing because, although they can contain several equilibria, all of them have equal rewards and are interchangeable. In this kind of MGs a Nash equilibrium corresponds to a worst-case scenario: if player 1 is playing an equilibrium strategy σ_1 then there is nothing that player 2 can do to improve its own payoff besides playing the corresponding strategy σ_2 and, because the game is zero-sum, there is no way player 1 can get a lower payoff than it is already receiving.

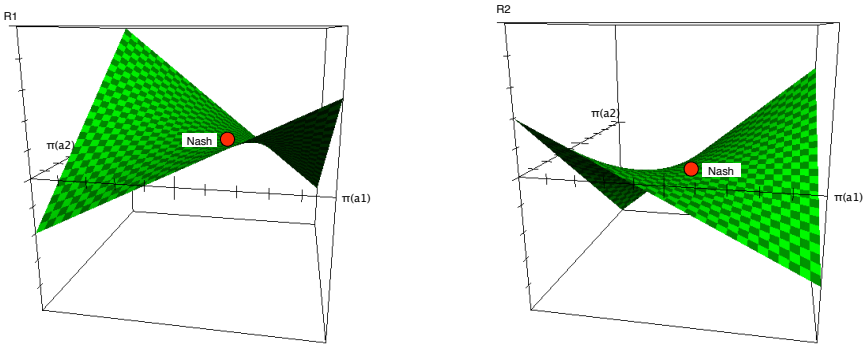
The equilibrium value is optimal in the sense that, for each of the players, the payoff will never be worst than that value. In alternated games the way of solving the game in the worst-case scenario is using a minimax approach where we maximize our reward given the other player is doing everything to minimize it; this procedure returns a deterministic strategy. However, in matrix games both players choose their actions at the same time and, in this situation, we cannot reason exactly like in alternated games.

However, a minimax operator that acts on the strategy space, rather than on the action space, can effectively find the equilibrium policy:

$$\max_{\sigma \in PD(A)} \min_{o \in O} \sum_{a \in A} \sigma(a) R(a, o)$$

where A represents player 1 action set, O represents player 2 (the opponent) action set and $R(a, o)$ the player 1 reward when joint action $\langle a, o \rangle$ is played. The optimal value will be a probability distribution function over the agent's actions instead of a singular action. This can be formulated as a linear programming problem which can be easily solved using a simplex algorithm. For further explanation on how to formulate the problem above as a linear program refer to (Owen, 1995) or (Littman, 1994).

In Figure 3.1 a general representation of the reward function for the zero-sum game in Table 3.4. In the graphic, the strategy is represented as the probability of choosing the first action (a_1 for player 1 and a_2 for player 2) because each player only has two actions and the probability of the second is inherently determined. Its possible to see that the equilibrium occurs at a saddle point of the reward function, which is common for zero-sum games.



(a) Reward function for player 1 (b) Reward function for player 2

Figure 3.1: Rewards and Nash equilibria for a zero-sum game

For this particular game the equilibrium occurs when $\pi_1 = \{0.540, 0.460\}$ and $\pi_2 = \{0.385, 0.615\}$.

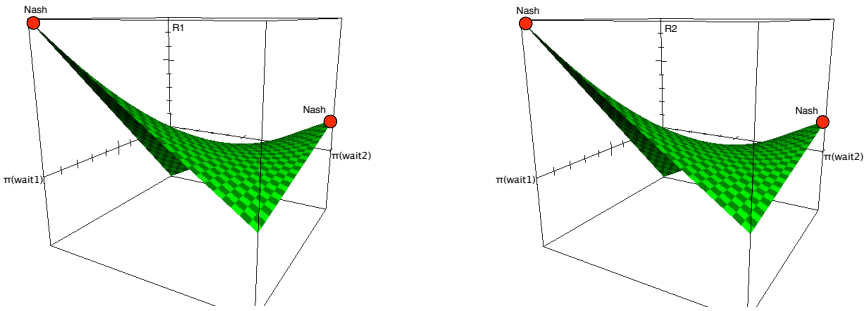
	a_2	b_2
a_1	(3 , -3)	(-5 , 5)
b_1	(-3 , 3)	(2 , -2)

Table 3.4: A zero-sum game

The second kind of games, called *team-games* also have trivial solutions: the payoffs are the same for all agents in the game and, in this situation, finding the a Nash equilibrium can be achieved just by finding the joint action corresponding to the higher payoff. As in MDPs, the strategy will be greedy, and if more than one greedy joint action exists, than all probability distributions over the greedy joint actions are Nash equilibria. Figure 3.2 represents the equilibria for a team game, particularly the one represented in Table 3.3.

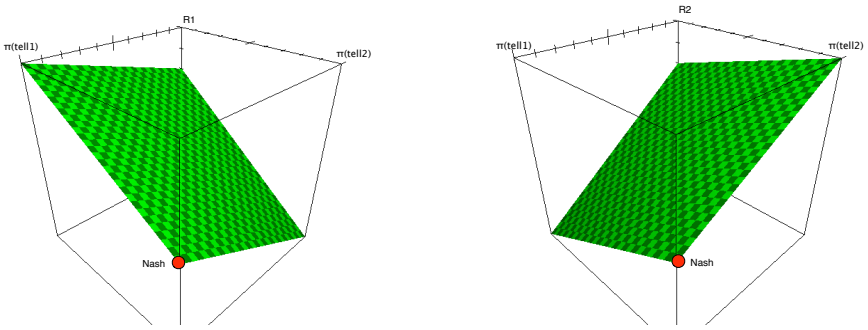
For general-sum games it is not so easy to compute the Nash equilibrium although a quadratic programming algorithm is enough when the game has only two-players as explained in (von Stengel, 1999; Koller *et al.*, 1996). Nevertheless, some games may have just one equilibrium, as is the case with the Prisoners Dilemma – the equilibrium is even deterministic for both players. The representation of the reward function for the Prisoners Dilemma matrices represents in Table 3.2 can be seen in Figure 3.3.

The kind of equilibrium represented in Figure 3.1 is generally called a *adversarial equilibrium*, which has the property that no player is hurt by



(a) Reward function for player 1 (b) Reward function for player 2

Figure 3.2: Rewards and Nash equilibria for a team game



(a) Reward function for player 1 (b) Reward function for player 1

Figure 3.3: Rewards and Nash equilibria for a version of the prisoner's dilemma

changes in all the other players strategies:

$$\forall_{\pi_{-i} \in PD(A_{-i})} \quad R_i(\langle \sigma_i^*, \sigma_{-i}^* \rangle) \leq R_i(\langle \sigma_i^*, \sigma_{-i} \rangle) \tag{3.5}$$

and, so, the equilibrium value acts as a worst-case optimal. The equilibria in zero-sum games are always adversarial equilibria, which occur at saddle points.

On the other hand, another important kind of equilibria are called a *coordination equilibrium*, which have the property of being maximal for all the agents:

$$\forall_{\pi_{-i} \in PD(A_{-i})} \quad R(\sigma_i^*) = \max_{a \in A} R_i(a) \tag{3.6}$$

Team-games always have at least one coordination equilibria – for Figure 3.2 it is the higher one, with a payoff of 4.

3.2 Stochastic Games

As above mentioned, *stochastic games* (SG) (Shapley, 1953) can be thought as an extension of matrix games and/or Markov decision processes in the sense that they deal with multiple agents in a multiple state situation. Formally, they can be defined as a tuple $(n, S, A_{1,\ldots,n}, T, R_{1,\ldots,n})$ where:

- n represents the number of agents
- S the state set
- A_i the action set of agent i and $A = A_1 \times \cdots \times A_n$ the joint action set.
- $T : S \times A \times S \to [0, 1]$ is a transition function which depends on the actions of all players.
- $R : S \times A \times S \to \mathcal{R}$ is a reward function representing the expected value of the next reward, which also depends on the actions of all players.

One can think of a SG as a succession of MGs, one for every state. In fact, SGs are a superset of the two frameworks previously presented – a MDP can be thought as a stochastic game when $n = 1$ and a MG can be thought as a stochastic game when $|S| = 1$.

The notion of policy can be extended from $MDPs$ and combined with the notion of strategy from matrix games. In fact, a policy $\pi_i : S \to PD(A_i)$ is still a collection of probability distributions over the available actions, one for each state, but now a different policy must be defined for each agent.

The collection of policies, one for each agent, is called a *joint policy* and it can be written $\pi = \langle \pi_i, \pi_{-i} \rangle$ where π_{-i} refers to a joint policy for all players except for i.

This notion of policy is also limited to Markovian policies. In fact, a SG is Markovian from the point of view of the game, although it is not Markovian from the point of view of each agent – if the others are using learning algorithms their policies are changing and, in this situation, the perceived behavior of the system is in fact changing. This is one of the facts that poses a greater challenge when trying to learn in stochastic games.

Optimality Concepts

As with MDPs, the goal of an agent in a stochastic game is to maximize its expected reward over time and, similarly to MDPs as well, the model for the expected reward over time has to be defined. Again, the most common model is the infinite-horizon discounted reward model, leading to the following definition of state values:

$$\begin{aligned} V_i^{\pi}(s) &= E\left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}^i \middle| s_t = s, \pi \right\} = \\ &= \sum_a \pi(s,a) \sum_{s'} T(s,a,s') \left[R_i(s,a,s') + \gamma V_i^{\pi}(s') \right] \end{aligned} \tag{3.7}$$

where r_{t+k+1}^i represents the immediate payoff received by agent i at time $t+k+1$ and $\pi(s,a)$ is the probability of choosing joint action a in state s . The big difference from this expression to the one from MDPs is the fact that the state values must defined for each agent but the expected value depends on the joint policy and not on the individual policies of the agents.

As for Q-values, they are defined as follows:

$$\begin{aligned} Q^{\pi}(s,a) &= E\left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s, a_t = a, \pi \right\} \\ &= \sum_{s' \in S} T(s,a,s') \big[R_i(s,a,s') + \gamma V_i^{\pi}(s') \big] \end{aligned} \tag{3.8}$$

and one thing that is visible from the equation is that Q-values also depend on the actions of all the players.

As in matrix games, the concept of optimal policy for one player of the game does not apply because the quality of a player's policy cannot be

computed independently of the policies of the other players. But it still makes sense to define the same kind of concepts as for matrix game, namely the concept of best-response function and the concept of Nash equilibrium.

In the context of SGs, a *best-response policy* for player i is one that is optimal with respect to some joint policy off the other players. The notation for that is:

$$\pi_i \in BR_i(\pi_{-i}) \tag{3.9}$$

and, as expected, $BR_i : S \times PD(A_i) \rightarrow 2^{S \times PD(A_i)}$ is called the best response function[3] for player i. $\pi_i^* \in BR_i(\pi_i)$ if and only if:

$$\forall \pi_{i \in S \times PD(A_i)}, \forall s \in S \quad V_i^{\langle \pi_i^*, \pi_{-i} \rangle}(s) \geq V_i^{\langle \pi_i, \pi_{-i} \rangle}(s) \tag{3.10}$$

A Nash equilibrium, in the context of stochastic games, is a collection of policies, one for each player, so that all of this policies are best-response policies and no player can do better by changing its policy. Formally:

$$\forall_{i=1 \ldots n} \quad \pi_i \in BR_i(\pi_{-i}) \tag{3.11}$$

The classification of SGs is similar to the one of MGs. The difference is that for a stochastic game to be a zero-sum game, all of the states must define a zero-sum matrix game and for it to be classified as a team game, all of the states must define team matrix games. The one that do not fall in any of these categories are generally called general-sum games.

Another class of stochastic games classic in game theory is *iterated games*. These are games with just one state (hence, only one reward matrix) but in which the game is repeated for a number of times or *ad eternum*. Some methods are concerned with learning in this type of games exactly.

Playing against stationary policies

When the policies of all but one of the agents are stationary, the stochastic game reduces to a MDP. In fact, all the other agents do is defining the transition probabilities and reward structure for the equivalent MDP. Supposing that only agent i is learning policy π_i and the joint policy of the other agents π_{-i} is fixed, the parameters of the MDP are:

- $S^{MDP} = S^{SG}$
- $A^{MDP} = A_i^{SG}$

[3] $S \times PD(A_i)$ represents the set of all possible markovian policies for player i and $2^{S \times PD(A_i)}$ represents its power set.

- $T^{MDP}(s, a_i, s') = \sum_{a_{-i} \in A^{SG}_{-i}} \pi_{-i}(s, a_{-i})\ T^{SG}(s, \langle a_i, a_{-i} \rangle, s')$
- $R^{MDP}(s, a_i, s') = \sum_{a_{-i} \in A^{SG}_{-i}} \pi_{-i}(s, a_{-i})\ T^{SG}(s, \langle a_i, a_{-i} \rangle, s')\ R^{SG}(s, \langle a_i, a_{-i} \rangle, s')$

In this situation, its clear a method like Q-learning or Sarsa will be enough to converge to the best-response (and optimal) policy to other players joint policy, without having to know the actions of the others at each step.

The next two sections will survey some algorithms that try to solve stochastic games. These can be divided in *best-response learners*, that try to learn a best-response policy to the joint policy of the other players, and *equilibrium learners*, that specifically try to learn equilibrium policies, regardless of the joint policy of the other agents.

3.3 Best-Response Learners

This class of methods are not specifically concerned with learning an equilibrium and just try to learn a policy that is optimal with respect to the policies of the other players. The advantage of not trying to learn the equilibrium is that the other players may be playing policies that are not best-response and, in this case, the agent can take advantage of that fact and try to obtain a higher return that the one that the equilibrium guarantees. However, against an agent without a stationary policy, or one that does not converge to a stationary policy, the methods may have trouble quickly adapting to what the other is doing and the optimal is no longer assured.

3.3.1 MDP methods

As it was observed previously, a stochastic game is equivalent to a Markov decision process from the point of view of one agent when the other agents are playing stationary policies. In this case, any algorithm that learns the optimal policy for a MDP is equally suited for a SG. Particularly, the algorithms for learning optimal policies presented in Chapter 2 are all suited for such a task. Some applications of MDP methods, like Q-learning, to a multi-agent domain have been proposed and not without success (Tan, 1993; Sen *et al.*, 1994).

Unfortunately, assuming that the other agents are not learning is not very realistic. Usually the other agents are also trying to maximize their reward and they cannot be exploited using deterministic policies, which are the ones most MDP methods find. Taking the matrix game of Table 3.4

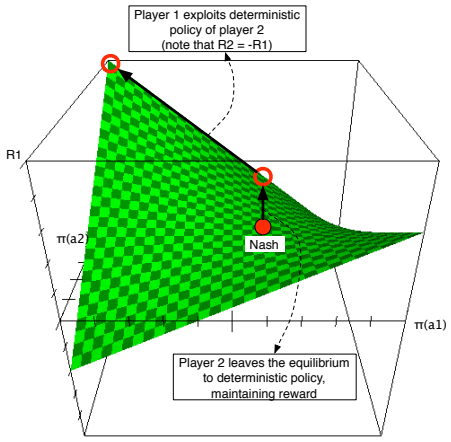


Figure 3.4: Deterministic policies can be exploited.

into account, if one agent (player 1) is playing the equilibrium strategy, the other (player 2) may play a deterministic strategy and get the same reward (which is equivalent to find a border solution for that player). However, once player 2 leaves the equilibrium, a learning player 1 can exploit that fact and play some policy which will lower the reward for player 2. Figure 3.4 represents the path taken.

3.3.2 Joint-action Learners

Joint-action learners (JALs) (Claus and Boutilier, 1998) are intended to solve iterated team matrix games, where the reward function is the same for all the involved agents and there is only one state which is played indefinitely. They differ from purely MDP methods in that they learn Q-values based on the joint-actions rather than just their own actions – it is assumed that they have full observability of the state and of the other agents' actions.

So, from the point of view of the learning method there is no difference between Q-learning and JALs, except for the action space. The problem arises when deciding which action to take: if the Q-functions for all the agents were coordinated, they would always choose the best action, when not performing exploratory moves. However, there is no guarantee that the other players are at the same learning stage, or even if they are learning at all. That is why (Claus and Boutilier, 1998) propose a value function to be

used instead of the Q-value, when deciding which action to take:

$$EV(a_i) = \sum_{a_{-i} \in A_{-i}} Q(\langle a_i, a_{-i} \rangle) \prod_{j \neq i} \widehat{\pi}_j(a_{-i}[j]) \tag{3.12}$$

where $\widehat{\pi}_j$ refers to a estimative of the policy being followed by player j. This estimator is obtained just by dividing the number of times agent j chose a given action by the number of trials.

3.3.3 Opponent Modeling

Opponent modeling (Uther and Veloso, 1997) is similar to JALs but it was initially applied to the context of zero-sum games. Like JALs, statistics of the number of visits to a state and the number of times an opponent chooses an action are maintained to obtain policy estimators for the other players. To be more precise, opponent modeling regards the other agents as one massive opponent with the ability to play joint actions and maintains statistics over them, which for the algorithm is in all equivalent. The estimator is:

$$\widehat{\pi_{-i}}(a_{-i}) = \frac{n(s, a_{-i})}{n(s)} \tag{3.13}$$

where $n(s, a_{-i})$ represents the number of times joint action (of the other players) π_{-i} has be chosen in state s and $n(s)$ represents the number of visits to a state.

The algorithmic version of opponent modeling can be seen in Algorithm 3.1.

3.3.4 WoLF Policy Hill Climber

Contrasting with the algorithms already shown, (Bowling, 2003; Bowling and Veloso, 2001) proposed the *WoLF policy hill climber* (WoLF-PHC) which can learn mixed policies and converges to a best-response strategy when all the other players also converge. Moreover, if all the players converge to best-response policies than those policies must form a Nash equilibrium.

The algorithm is on-policy and works by modifying the policy directly according to the expected reward values it maintains. The idea of policy hill climbers (PHC) has been present in some reinforcement methods but, nevertheless, the novelty of the work of (Bowling, 2003) is that a variable learning rate is combined with the PHC – WoLF stands for *Win or Learn Fast*. To quantify the quality of the current policy, an average policy is also

Algorithm 3.1 Opponent modeling

Initialize $Q(s, a)$ arbitrarily
$\forall_{s \in S} \forall_{a_{-i} \in A_{-i}} \quad n(s) \leftarrow 0$ and $n(s, a_{-i}) \leftarrow 0$

Initialize s
loop
$a_i \leftarrow$ probabilistic outcome of policy (e.g. ϵ-greedy) based on $O(s, a_i)$
with $O(s, a_i) = \sum_{a_{-i}} \frac{n(s,a_{-i})}{n(s)} Q(s, \langle a_i, a_{-i} \rangle)$

Take action a_i, observe reward r, next state s' and other players joint action a_{-i}

$Q(s, \langle a_i, a_{-i} \rangle) \leftarrow Q(s, \langle a_i, a_{-i} \rangle) + \ \alpha\big(r + \gamma V(s') - Q(s, \langle a_i, a_{-i} \rangle)\big)$
with $V(\sigma) = \max_{a_i} \sum_{a_{-i}} \frac{n(s,a_{-i})}{n(s)} Q(s, \langle a_i, a_{-i} \rangle)$

$n(s, a_{-i}) \rightarrow n(s, a_{-i}) + 1$
$n(s) \rightarrow n(s) + 1$

$s \leftarrow s'$
end loop

maintained and the current policy expected value is compared to the one of the average policy.

Suppose the agent is following policy π and has an average policy of $\tilde{\pi}$, the agent is said to be wining if and only if:

$$\sum_{a'} \pi(s, a')Q(s, a') > \sum_{a'} \tilde{\pi}(s, a')Q(s, a')$$

The average policy is calculated by maintaining a statistic of visits to a given state and using it to update the policy.

In Algorithm 3.2 the algorithm for WoLF-PHC is shown. Although some interesting results were given in (Bowling, 2003) and the method performs much faster than its fixed learning rate counterparts, its convergence is yet to be formally proved.

3.4 Equilibrium Learners

Equilibrium Learners specifically try to find policies which are Nash equilibria for the stochastic game. Usually, as it is hard to find such equilibria, they focus on a smaller class of problems, for example zero-sum games or two-person general-sum. The advantage of finding the Nash equilibrium is that the agent learns a lower bound for performance and, in this situation, it becomes fairly independent of the policies being played by the other agents – it will get at least the amount of return which corresponds to the equilibrium.

The general idea is to find an equilibrium strategy[4] for each state of the game and composing all the strategies to find the Nash policy.

In fact, a Nash equilibrium policy in a stochastic game can always be reduced to a collection of Nash equilibrium strategies, one for each state of the game. So, for an equilibrium policy π^* and a state s, the matrix game whose equilibrium is the strategy $\pi^*(s)$ can be defined by the following rewards:

$$R_i(a) = Q_i^{\pi^*}(s, a) \tag{3.14}$$

Generally, a solution for an equilibrium learner would be a fixed point in π^* of the following system of equations:

$$\forall_{i=1\ldots n} \quad Q_i^*(s, a) = \sum_{s' \in S} R_i(s, a, s') + \gamma T(s, a, s')V_i^{\pi^*}(s') \tag{3.15}$$

[4] A strategy is nothing more than a policy for just one state.

Algorithm 3.2 WoLF Policy Hill Climbing

Initialize $Q(s,a)$ and π arbitrarily (e.g. $\pi(s,a) \rightarrow \frac{1}{|A_i|}$)
$\forall_{s \in S} \;\; n(s) \leftarrow 0$.

Initialize s
loop
$a \leftarrow$ probabilistic outcome of policy $\pi(s)$ {Mixed with exploration policy}

Take action a, observe reward r and next state s'
$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)$

Update average policy $\tilde{\pi}$:

$$n(s) \leftarrow n(s) + 1$$

$$\tilde{\pi}(s) \leftarrow \tilde{\pi}(s) + \frac{1}{n(s)} \left(\pi(s) - \tilde{\pi}(s)\right)$$

if $\sum_{a'} \pi(s,a')Q(s,a') > \sum_{a'} \tilde{\pi}(s,a')Q(s,a')$ **then**
$\delta \leftarrow \delta_w$ (winning)
else
$\delta \leftarrow \delta_l$ (loosing)
end if

Update policy $\pi(s)$:

$$\delta_{sa} = \min\left(\pi(s,a), \frac{\delta}{|A_i| - 1}\right)$$

$$\Delta_{sa} = \begin{cases} -\delta_{sa} & a \neq \arg\max_{a'} Q(s,a') \\ \sum_{a' \neq a} \delta_{sa'} & otherwise \end{cases}$$

$$\pi(s,a) \leftarrow \pi(s,a) + \Delta_{sa}$$

Normalize $\pi(s)$.

$s \leftarrow s'$
end loop

where $V_i^{\pi^*}(s')$ represents the equilibrium value for agent i when the joint-policy being played is the Nash equilibrium π^* and is computed with respect to the Q-values. This is similar to a Bellman optimality equation except for the way the state value function is computed. In fact, the Q-function could be estimated through a stochastic approximation procedure very similar to the one of Q-learning:

$$\forall_{i=1\ldots n} \quad Q_i(s,a) \rightarrow Q_i(s,a) + \alpha\big(r_i + \gamma V_i(s') - Q_i(s,a)\big) \tag{3.16}$$

with $V_i(s')$ being the value of a Nash equilibrium policy for agent i. The big problem with this approach is that, unlike what happens in MDPs, there is no guarantee that Equation 3.15 has only one fixed point. In fact, generally there are several Nash equilibria in a stochastic game, which results in the fact that there is no unique way of computing a Nash equilibrium value for each matrix game defined by $Q(s,a)$. Usually, this approach works when considering types of games with just one equilibria or, if that is not the case, some coordination device is necessary so that the players all converge to the same equilibrium.

There is an implicit assumption of full observability in this approach – each agent has to store Q-values for all the joint actions.

The general algorithmic version of this procedure can be seen in 3.3.

Algorithm 3.3 General equilibrium learner algorithm

Initialize $Q(s,a)$ arbitrarily

Initialize s
loop
$a_i \leftarrow$ probabilistic outcome of Nash policy derived from $Q(s,a)$, for player i {Mixed with exploration policy}

Take action a_i, observe reward r, next state s' and the joint action of other players a_{-i}

for $i = 1 \ldots n$ **do**
$Q_i(s,\langle a_i, a_{-i}\rangle) \leftarrow Q_i(s,\langle a_i, a_{-i}\rangle) + \alpha\big(r_i + \gamma V_i(s') - Q_i(s,\langle a_i, a_{-i}\rangle)\big)$
end for
where $V(s) = Nash\left([Q(s,a)]\right)$

$s \leftarrow s'$
end loop

3.4.1 Minimax-Q

The method of *Minimax-Q* (Littman, 1994) is designed to work with zero-sum stochastic games. In zero-sum games there is only one equilibrium and, as stated previously, it can be found using linear programming. In this situation, the state value can be univocally computed by the minimax procedure and Equation 3.15 only has one fixed point. It is expected that Algorithm 3.3 converges to the equilibrium policy. In fact, this has been proved by (Littman and Szepesvári, 1996).

Using minimax to compute the Nash equilibrium, Equation 3.15 can be used to describe the Nash condition and Equation 3.16 can be used as an update rule if the Nash value is computed like

$$V(s) = \max_{\pi \in PD(A)} \min_{o \in O} \sum_{a \in A} \pi(s,a) Q(s, \langle a, o \rangle) \tag{3.17}$$

where A represents the action space of the learner and O the action space of the opponent. There is no need for considering a two-dimensional state function because, as the rewards are always symmetric for the agents, so will be the state values and Q-values.

The algorithm can be particularized from Algorithm 3.3 to the Minimax-Q form of Algorithm 3.4.

Algorithm 3.4 Minimax-Q learner

Initialize $Q(s, \langle a, o \rangle)$ and $\pi(s)$ arbitrarily

Initialize s
loop
  $a \leftarrow$ probabilistic outcome of $\pi(s)$ {Mixed with exploration policy}

  Take action a, observe reward r, next state s' and opponent action o

  $Q(s, \langle a, o \rangle) \leftarrow Q(s, \langle a, o \rangle) + \alpha\big(r + \gamma V(s') - Q(s, \langle a, o \rangle)\big)$
  with $V(s) = \max_{\pi' \in PD(A)} \min_{o' \in O} \sum_{a' \in A} \pi(s, a')\, Q(s, \langle a', o' \rangle)$
  $\pi(s) \rightarrow \arg \max_{\pi' \in PD(A)} \min_{o' \in O} \sum_{a' \in A} \pi(s, a')\, Q(s, \langle a', o' \rangle)$

  $s \leftarrow s'$
end loop

3.4.2 Nash-Q

Nash-Q (Hu and Wellman, 1998; Hu and Wellman, 2004) tries to address the general problem of learning in two-player general-sum games, using the algorithm structure presented in Algorithm 3.3. The equilibrium value is computed using a quadratic programming approach, suited for computing Nash equilibria in general-sum games. Nevertheless, the recurrent problem of not knowing which equilibrium to choose is patent here. To avoid this problem, two strict assumptions are devised:

1. All intermediate games over the Q-values must have only one equilibrium.
2. The equilibrium in each game must either be a saddle point (in adversarial situations from which zero-sum games are a particular case) or a global maximum, maximizing each agent's reward.

This conditions are often too strict because it is not possible to predict wether they still hold while learning. The convergence difficulties of the method have been clarified by (Bowling, 2000)

3.4.3 Friend-or-Foe-Q

Motivated by the assumptions of Nash-Q, (Littman, 2001) extended the original algorithm Minimax-Q to solve a more general class of stochastic games. In each state, the method is told wether the agent is playing with a *Friend*, and the Nash would be a coordination equilibria and a global optimum, or against a *Foe*, with the game having an adversarial equilibrium in a saddle point. It is proved that:

- If the game has at least one coordination equilibria, then all coordination equilibria have the same value.
- If the game has at least one adversarial equilibria, then all adversarial equilibria have the same value.

The value function is computed differently wether the algorithm is told to be playing against a foe, and uses minimax:

$$V(s) = \max_{\pi \in PD(A)} \min_{o \in O} \sum_{a \in A} \pi(s,a) Q(s, \langle a, o \rangle) \tag{3.18}$$

or with a friend, and uses max:

$$V(s) = \max_{a \in A, o \in O} Q(s, \langle a, o \rangle) \tag{3.19}$$

For simplicity, only the two-person game is being considered, although the algorithm could extend to a more general situation.

The algorithm version of *Friend-or-Foe-Q* can be seen in Algorithm 3.5.

Algorithm 3.5 Friend-or-Foe-Q learner

Initialize $Q(s, \langle a, o \rangle)$ and $\pi(s)$ arbitrarily

Initialize s
loop
$a \leftarrow$ probabilistic outcome $\pi(s)$ {Mixed with exploration policy}

Take action a, observe reward r, next state s' and opponent action o

$Q(s, \langle a, o \rangle) \leftarrow Q(s, \langle a, o \rangle) + \alpha\big(r + \gamma V(s') - Q(s, \langle a, o \rangle)\big)$
where
if Playing against foe **then**
$V(s) = \max_{\pi' \in PD(A)} \min_{o' \in O} \sum_{a' \in A} \pi(s, a')\, Q(s, \langle a', o' \rangle)$
$\pi(s) \rightarrow \arg \max_{\pi' \in PD(A)} \min_{o' \in O} \sum_{a' \in A} \pi(s, a')\, Q(s, \langle a', o' \rangle)$
else
$V(s) = \max_{a' \in A, o' \in O} Q(s, \langle a', o' \rangle)$
$\pi(s, a) = \begin{cases} 1 & a = \arg \max_{a' \in A} \{ \max_{o' \in O} Q(s, \langle a', o' \rangle) \} \\ 0 & \text{otherwise} \end{cases}$
end if

$s \leftarrow s'$
end loop

The convergence of the method was proved by (Littman, 2001) which also discussed that one limitation is, when playing with a friend, coordinating to choose one of the possible coordination equilibria. Its easy to see, by looking at the game of Table 3.5, a variation of the game of Table 3.3 which is not fully cooperative, that there are two coordination equilibria which occur at $\{Wait, Go\}$ and $\{Go, Wait\}$. However, if players are not coordinated they may play $\{Go, Go\}$ or $\{Wait, Wait\}$, leading to lower rewards than on the equilibrium.

	Wait	Go
Wait	(0 , 0)	(4 , 2)
Go	(4 , 2)	(-2 , -2)

Table 3.5: A matrix game with two equilibria.

Chapter 4

Conclusions and Research Directions

Concerning the single agent framework, this work tried to survey some key algorithms and the main theoretical ideas that act as a foundation for most of the reinforcement learning body of knowledge. All the temporal difference algorithms addressed are well established methods whose convergence properties have been proved.

As for the extension to more than one step updates, using eligibility traces has proved to accelerate the learning, when compared with the one step counterparts, although the convergence results have been only partially proved.

Some other directions could be addressed as they are fundamental in bringing reinforcement learning closer to the real and feasible applications.

Generalization and Function Approximation. The algorithms addressed here can be generally called *tabular* as they assume the values are stored in table, which makes sense for finite MDPs. However, in most applications the state space is not finite (usually continuous) or the memory available, and the time to process it, are serious limitations of the system.

In this case, the value functions have to be generalized for other inputs of state and action – a close complementary relation arises between reinforcement learning and *supervised learning* algorithms. Popular example are the use of neural networks, e.g.. (Jin, 1993; Tesauro, 1992), cerebellar model articulator controllers (CMAC), as in (Watkins, 1989), SVNs, among many others.

Model-based methods. Sometimes it can be a good idea to have a explicit model of the process and at the same time maintain the values necessary to obtain the policy. The idea is that simulated experiments can be generated using the model and provided to the learning algorithm, causing the algorithm to converge must faster, although biased by the correctness of the model. In a sense, this works as an integration of planning, acting and learning.

Methods to integrate planning, learning and acting include:

Certainty equivalence, (Kumar and Varaiya, 1986) in which the parameters of the MDP are learned previously, through exploring the environments and storing statistics over transitions and rewards, and then a dynamic programming algorithm is applied to obtain optimal values and functions. This was also the approach of (Neto and Lima, 2005), based on Minimax-Q, for learning a model and optimal policies for the multi-agent problem of zero-sum games.

Dyna-Q, (Sutton, 1991) simultaneously builds a model from experience, uses the experience to adjust the policy and randomly generates simulations from the model to adjust the policy. This architecture proved to perform much well then the naive model-based method referred previously.

Prioritized Sweeping, (Moore and Atkeson, 1993) uses a similar approach of Dyna-Q but, instead of generating simulations randomly, it keeps a priority for each state and generates simulations according to this priority.

Hierarchical Reinforcement Learning. Creating hierarchies of learners is a common approach to reducing the complexity of the system and concentrating on smaller and local problems. Although the solutions may be sub-optimal, they are much more feasible and take less time to obtain than those that consider all the state.

In (Barto and Mahadevan, 2003) the main hierarchical models are analyzed, from the perspective of *Semi-MDPs* (Howard, 1971). Those are considered to be the frameworks of *options* (Sutton *et al.*, 1999), *MAXQ* (Dietterich, 2000) and *hierarchies of abstract machines* (HAMs) (Parr and Russel, 1997).

Partially Observable Markov Decision Processes. An important question on using reinforcement algorithms to tackle real world problems is to consider, on the models, the question of partial observability. The framework of *partially observable Markov decision processes* extends the regular MDPs to incorporate beliefs about the state. The methods then start working on beliefs rather than on states directly.

(Cassandra, 1994; Littman *et al.*, 1995) apply dynamic programming algorithms to solve POMDPs. (Hansen *et al.*, 2004) tries to address the problem of partial observability on stochastic games. An interesting approach is that of (Spaan and Vlassis, 2004) where the belief space is sampled and methods work on the samples, rather than on the space directly.

Concerning the multi agent framework, the most significant body of work can be divided into two separate kinds of methods: best-response learners and equilibrium learners.

The first kind of algorithms has the advantage of trying to exploit the environment in the best way it can. In fact, if the other agents are not playing Nash equilibria, best-response learners might end-up receiving higher payoffs than if they had played equilibrium policies. From the algorithms presented, WoLF-PHC is perhaps the best suited for learning in general-sum stochastic games for two reasons: it converges (the experiments seem to indicate it) to a stochastic policy, which can be a Nash equilibrium if the other agents are all best-response learners, and it has a variable learning rate, which can turn the policy from a softer to a harder one (or vice-versa), indirectly controlling the focus from exploration to exploitation.

As for the second kind of methods, their advantage is in that they provide a solid way of finding policies which give some performance guarantees. Although they can not exploit the weaknesses of the opponents, the policies learned provide a lower bound for the expected return. The problem with equilibrium learners is that SGs sometimes have several equilibria and it is not trivial to know which equilibrium to choose. In this context, Minimax-Q is the most consistent method because it works well in the domain it is restricted to. Nash-Q is a general and interesting method but imposes strict and non-trivial conditions on the initial problem it proposes to solve. Friend-or-Foe-Q is a generalization of Minimax-Q but has the problem of needing and oracle to coordinate between equilibria when learning with Friends. (Shoham *et al.*, 2004) criticize the equilibrium approach and argue that using learning for action coordination in multi-agent systems stops making sense when some external mechanism has to be used to coordinate between equilibria.

As for research directions in multi-agent learning, there are some interesting open topics that remain to be addressed in detail.

Local best-response policies. All the algorithms presented try to address the problem of multi-agent learning by either being optimal (in the best-response sense) to the other agents actions or playing a Nash equilibrium. However, those two objectives might be to hard to tackle for a system with limited resources or when the number of agents is to high.

An alternative is to aim for global sub-optimal goals by playing best-response or Nash locally – in large adversarial team situations, each agent is typically concerned with the agents around him, wether in his team or in another team. (Kok *et al.*, 2005) addresses that problem by using predefined coordination graphs and, in this case, each agent just has to learn to coordinate optimally with the agents defined in the graph – the optimal coordination is lost but the method becomes feasible.

Beliefs about other agents. (Chang and Kaelbling, 2001) discuss a classification for multi-agent learning systems based not only on their policies but on their beliefs about other agents policies. They argue that those kind of agents might take advantage of best-response learners by leading them to converge to a situation where they can be exploited and then shifting the learning mechanism to take advantage of that. They come up with an algorithm named *PHC-exploiter* which tries to exploit a PHC learner in the way described. The algorithm converges gradually to a cycle of winning/loosing but which eventually has a positive average reward for zero-sum games.

The interesting thing about this approach is that it extends the idea of converging to a stationary policy and, instead, goes further by leading the system to a policy cycle which, in the end, does well on average.

Domain dependent algorithms. The analysis of the convergence problems of Nash-Q gives some insight on the difficulty of designing an algorithm to behave well in all kinds of general-sum games. Another approach is designing hybrid algorithms that detect (or are informed) the type of game being played and use a different learning method. Friend-or-Foe-Q approached that same idea, which is also the one taken by (Powers and Shoham, 2004).

Multi-robot systems. The multi-agent perspective is no doubt interesting from theoretical and even practical aspects. However, their integration

in the growing field of multi-robot systems is, most of the times, not straightforward because robots pose additional problems due to their nature – they often carry a bag of coupled problems which are not trivial to tackle individually. Its then a great challenge by itself to make multi-agent learning methods applicable to robots, and one from which lots of interesting technologies will surely stem. In this context, (Yang and Gu, 2004) survey learning techniques and discuss their applicability to multi-robot systems.

The field of multi-agent learning, particularly its applicability to multi-robot systems, is still very open with many interesting questions to be answered. It was in this perspective that this work, far from trying to do a comprehensive survey of multi-agent learning research, instead tried to address the main results in single and multi-agent reinforcement learning. To us, it served very well the purpose of consolidating basic concepts on the field and expanding our view of the main challenges and research directions to be pursued and, hopefully, it will be useful to someone else in the same manner.

Bibliography

Barto, Andrew G. and Sridhar Mahadevan (2003). Recent advances in hierarchical reinforcement learning. *Special Issue on Reinforcement Learning, Discrete Event Systems Journal* pp. 41–77.

Bellman, Richard Ernest (1957). *Dynamic Programming*. Princeton Press.

Bertsekas, Dimitri P. (1982). Distributed dynamic programming. *IEEE Transactions on Automatic Control* **27**, 610–616.

Bertsekas, Dimitri P. (1983). Distributed asynchronous computation of fixed points. *Mathematical Programming* **27**, 107–120.

Bertsekas, Dimitri P. (1987). *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall.

Bertsekas, Dimitri P. (1995). *Dynamic Programming and Optimal Control*. Vol. 1 and 2. Athena Scientific.

Bertsekas, Dimitri P. and John N. Tsitsiklis (1989). *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall.

Bertsekas, Dimitri P. and John N. Tsitsiklis (1996). *Neuro-Dynamic Programming*. Athena Scientific.

Bhulai, Sandjai (2002). Markov Decision Processes: the control of high-dimensional systems. PhD thesis. Vrije Universiteit.

Bowling, Michael (2000). Convergence problems of general-sum multiagent reinforcement learning. In: *Proceedings of the Seventeenth International Conference on Machine Learning*. Morgan Kaufman. pp. 89–94.

Bowling, Michael (2003). Multiagent Learning in the Presence of Agents with Limitations. PhD thesis. Carnegie Mellon University. Pittsburgh.

Bowling, Michael and Manuela Veloso (2001). Rational learning of mixed equilibria in stochastic games. In: *Proceedings of the Seventeenth Joint Conference on Artificial Intelligence*.

Cassandra, Anthony R. (1994). Optimal policies for partially observable markov decision processes. Technical report. Department of Computer Science, Brown University. Providence.

Cassandras, Christos G. and Stéphane Lafortune (1999). *Introduction to Discrete Event Systems*. Kluwer Academic Publishers. Boston.

Chang, Yu-Han and Leslie Pack Kaelbling (2001). Playing is believing: the role of beliefs in multi-agent learning. In: *Advances in Neural Information Processing Systems*. Vancouver.

Claus, Caroline and Craig Boutilier (1998). The dynamics of reinforcement learning in cooperative multiagent systems. In: *Proceedings of the Fifteenth National Conference on Artificial Intelligence*. AAAI Press.

Dayan, Peter (1992). The convergence of td(λ) for general λ. *Machine Learning* **8**, 341–362.

Dayan, Peter and Terrence J. Sejnowski (1994). TD(λ) converges with probability 1. *Machine Learning* **14**(3), 295–301.

Dietterich, Thomas G. (2000). Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research* **13**, 227–303.

Hansen, Eric A., Daniel S. Bernstein and Shlomo Zilberstein (2004). Dynamic programming for partially observable stochastic games. In: *Proceedings of the Nineteenth National Conference on Artificial Intelligence*. San Jose.

Howard, Richard Author (1960). *Dynamic Programming and Markov Decision Processes*. MIT Press.

Howard, Richard Author (1971). *Dynamic Probabilistic Systems: Semi-Markov and Decision Processes*. Wiley.

Hu, Junling and Michael P. Wellman (1998). Multiagent reinforcement learning: Theoretical framework and an algorithm. In: *Proceedings of the Fifteenth International Conference on Machine Learning*. Morgan Kaufman. pp. 242–250.

Hu, Junling and Michael P. Wellman (2004). Nash Q-learning for general-sum stochastic games. *Journal of Machine Learning Research* **4**, 1039–1069.

Jin, Long-Li (1993). Reinforcement Learning for Robots using Neural Networks. PhD thesis. Carnegie Mellon University.

Kok, Jelle R., Matthijs T. J. Spaan and Nikos Vlassis (2005). Non-communicative multi-robot coordination in dynamic environments. *Robotics and Autonomous Systems* **50(2-3)**, 99–114.

Koller, Daphne, Nimrod Meggido and Bernhard von Stengel (1996). Efficient computation of equilibria for extensive two-person games. *Games and Economic Behavior* **14**, 247–259.

Kumar, P. R. and P. P. Varaiya (1986). *Stochastic Systems: Estimation, Identification and Adaptive Control*. Prentice-Hall.

Littman, Michael L. (1994). Markov games as a framework for multi-agent reinforcement learning. In: *Proceedings of the Thirteenth International Conference on Machine Learning*. New Brunswick. pp. 157–163.

Littman, Michael L. (2001). Friend-or-foe Q-learning in general-sum games. In: *Proceedings of the Eighteenth International Conference on Machine Learning*. Williamstown. pp. 322–328.

Littman, Michael L. and G. Szepesvári (1996). A generalized reinforcement learning model: Convergence and applications. In: *Proceeding of the 13th International Conference on Machine Learning*. Morgan Kaufman. pp. 310–318.

Littman, Michael L., Anthony R. Cassandra and Leslie Pack Kaelbling (1995). Efficient dynamic-programming updates in partially observable markov decision processes. Technical report. Department of Computer Science, Brown University. Providence.

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

Neto, Gonçalo and Pedro Lima (2005). Minimax value iteration applied to robotic soccer. In: *IEEE ICRA 2005 Workshop on Cooperative Robotics*. Barcelona, Spain.

Owen, Guillermo (1995). *Game Theory*. 3rd ed.. Academic Press.

Parr, Ronald and Stuart Russel (1997). Reinforcement learning with hierarchies of machines. In: *NIPS 97*.

Peng, Jing (1993). Efficient Dynamic Programming-Based Learning for Control. PhD thesis. Northeastern University. Boston.

Peng, Jing and Ronald J. Williams (1996). Incremental multi-step Q-learning. *Machine Learning* **22**, 283–290.

Powers, Rob and Yoav Shoham (2004). New criteria and a new algorithm for learning in multi-agent systems. In: *Proceedings of NIPS 04/05*.

Puterman, Martin L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley-Interscience.

Ross, Sheldon M. (1983). *Introduction to Stochastic Dynamic Programming*. Academic Press.

Rummery, G. A. (1995). Problem Solving with Reinforcement Learning. PhD thesis. Cambridge University.

Rummery, G. A. and M. Niranjan (1994). On-line q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR166. Cambridge University.

Sen, Sandip, Mahendra Sekaran and John Hale (1994). Learning to coordinate without sharing information. In: *Proceedings of the 13th National Conference on Artificial Intelligence*.

Shapley, L. S. (1953). Stochastic games. In: *Classics in Game Theory*. Princeton University Press.

Shoham, Yoav, Rob Powers and Trond Grenager (2004). On the agenda(s) of research on multi-agent learning. In: *2004 AAAI Fall Symposium on Artificial Multi-Agent Learning*.

Singh, Satinder P., Tommi Jaakkola, Michael L. Littman and Csaba Szepesvári (2000). Convergence results for single-step on-policy reinforcement learning algorithms. *Machine Learning* **38**, 287–308.

Spaan, Matthijs T. J. and Nikos Vlassis (2004). A point-based POMDP algorithm for robot planning.. In: *Proceedings of the IEEE International Conference on Robotics and Automation.*. New Orleans, Louisiana.

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

Sutton, Richard S. (1991). Planning by incremental dynamic programming. In: *Proceedings Ninth International Conference on Machine Learning*. Morgan Kaufmann. pp. 353–357.

Sutton, Richard S. (1996). Generalization in reinforcement learning: Successful examples using sparse coding. In: *Advances in Neural Information Processing Systems 8*.

Sutton, Richard S. and Andrew G. Barto (1998). *Reinforcement Learning*. MIT Press.

Sutton, Richard S., Doina Precup and Satinder P. Singh (1999). Between MDPs and semi-MDPs: a framework for temporal abstraction in reinforcement learning. *Artificial Intelligence* **112**, 181–211.

Tan, Ming (1993). Multi-agent reinforcement learning: Independent vs. cooperative games. In: *Tenth International Conference on Machine Learning*. Amherst.

Tesauro, Gerald (1992). Practical issues in temporal difference learning. In: *Advances in Neural Information Processing Systems* (John E. Moody, Steve J. Hanson and Richard P. Lippmann, Eds.). Vol. 4. Morgan Kaufmann Publishers, Inc.. pp. 259–266.

Thrun, Sebastian B. (1992). The role of exploration in learning control. In: *Handbook of Intelligent Control*. Van Nostrand Reinhold.

Tsitsiklis, John N. (1994). Asynchronous stochastic approximation and Q-learning. *Machine Learning* **16**, 185–202.

Uther, William and Manuela Veloso (1997). Adversarial reinforcement learning. Technical Report CMU-CS-03-107. Carnegie Mellon University.

von Neumann, John and Oskar Morgenstern (1947). *Theory of Games and Economic Behavior*. Princeton University Press. Princeton.

von Stengel, Bernhard (1999). Computing equilibria for two-person games. In: *Handbook of Game Theory* (Robert J. Aumann and Sergiu Hart, Eds.). Vol. 3. Chap. 45. North-Holland. Amsterdam.

Watkins, Christopher J. C. H. (1989). Learning from Delayed Rewards. PhD thesis. King's College. Cambridge, UK.

Watkins, Christopher J. C. H. and Peter Dayan (1992). Technical note: Q-learning. *Machine Learning* **8**, 279–292.

Yang, Erfu and Dongbing Gu (2004). Multiagent reinforcement learning for multi-robot systems: A survey. Technical Report CSM-404. University of Essex.