Learning and Generalization in Atari Games

Daniel Furelos Blanco
Supervised by Anders Jonsson

July 2015

Bachelor of Computer Science

Department of Information and Communication Technologies
To my family
I would like to express my very great appreciation to my supervisor, Anders Jonsson, for his advices, patience and honest opinions during the project.

I would also like to offer my special thanks to my fellows in The Binary Three team: Adrià Garriga and Sergio Almendros. Both of them have been incredible sources of support during the project by sharing endless hours in the library, and also for providing me with extra resources to finish the project earlier than expected.

My special thanks are extended to my main teammates during the degree (alphabetically): Bartomeu Galmés, Alex Gonzalo, Ignacio Granados, Álvaro Ibáñez, Àlex Juez and Víctor López. Together with Adrià and Sergio, they have proved to be wonderful people to work with. They deserve the best.

I wish to express my sincere thanks to the developers of the Stella and the Arcade Learning Environment. Without their work, this thesis could not have been possible. Also I would to thank people at Keonn Technologies for understanding how important this thesis is for me.

Most importantly, I am very grateful to my parents and my brother, for all their support in everything I do.
Abstract

This thesis describes the design of agents that learn to play Atari games using the Arcade Learning Environment (ALE) framework to interact with them. The application of machine learning in video games, given its high complexity, is considered to be a bridge towards real-world domains such as robotics.

The goal in Atari games is to achieve the highest possible score. To solve this task, reinforcement learning and search techniques are used. These algorithms outperform humans in 30 of the 61 games supported by ALE.

Since humans are very good at making generalizations between games, special emphasis is given to evaluating how well an agent learns from multiple games simultaneously. These experiments usually result in a higher score for specific pairs of games. Besides, there are games that tend to increase their score when playing with other games, whereas there are games that help others to perform better.
Resum

Aquesta tesi descriu el disseny d’agents que aprenen a jugar a jocs d’Atari utilitzant el framework Arcade Learning Environment (ALE) per a interactuar amb ells. L’aplicació d’aprenentatge automàtic en videojocs, donada la seva alta complexitat, es considera un pont cap a dominis com la robòtica.

L’objectiu als jocs d’Atari és aconseguir la major puntuació possible. Per a resoldre aquesta tasca, s’utilitzen tècniques d’aprenentatge per reforç i cerca. Aquests algoritmes superen als humans en 30 dels 61 jocs suportats per ALE.

Com els humans són molt bons fent generalitzacions entre jocs, es fa especial èmfasi en avaluar com un agent pot aprendre de múltiples jocs jugats simultàniament. Aquests experiments solen resultar en una major puntuació per a parelles específiques de jocs. A més, hi ha jocs que tendeixen a incrementar la seva puntuació quan juguen amb altres, mentre que també hi ha jocs que ajuden a altres a actuar millor.
Resumen

Esta tesis describe el diseño de agentes que aprenden a jugar a juegos de Atari usando el framework Arcade Learning Environment (ALE) para interactuar con ellos. La aplicación de aprendizaje automático en videojuegos, dada su alta complejidad, se considera un puente hacia dominios como la robótica.

El objetivo en los juegos de Atari es conseguir la mayor puntuación posible. Para resolver esta tarea, se utilizan técnicas de aprendizaje por refuerzo y búsqueda. Estos algoritmos superan a los humanos en 30 de los 61 juegos soportados por ALE.

Como los humanos son muy buenos haciendo generalizaciones entre juegos, se hace especial énfasis en evaluar cómo un agente puede aprender de múltiples juegos jugados simultáneamente. Estos experimentos suelen resultar en una mayor puntuación para pares específicos de juegos. Además, hay juegos que tienden a incrementar su puntuación cuando juegan con otros, mientras que también hay juegos que ayudan a otros a actuar mejor.
Contents

Acknowledgements v

Abstract vii

Resum viii

Resumen ix

List of Figures xiii

List of Tables xv

Abbreviations xvii

1 Introduction 1
  1.1 Context ................................................. 1
  1.2 Problem ............................................... 3
  1.3 Thesis Organization ................................. 5

2 Background Theory 7
  2.1 Atari 2600 ............................................. 7
  2.2 Sequential Decision Making ......................... 9
  2.3 Markov Decision Processes .......................... 11
  2.4 Reinforcement Learning ................................ 13
    2.4.1 The Generalized Policy Iteration Pattern .......... 13
    2.4.2 Function Approximation ............................ 15
      2.4.2.1 Gradient-Descent Methods ....................... 16
      2.4.2.2 Linear Function Approximation .................. 17
    2.4.3 Sarsa ............................................... 18
    2.4.4 Optimistic Initialization .......................... 20
  2.5 Monte Carlo Tree Search (MCTS) ..................... 21
# Contents

2.5.1 General Approach ........................................... 22
2.5.2 Characteristics ............................................. 24
2.5.3 The UCT Algorithm ......................................... 25
2.5.4 All Moves As First (AMAF) ................................. 28
2.6 Arcade Learning Environment (ALE) .......................... 30

3 Methodology ......................................................... 33
3.1 Programming Language and Implementation Method ........... 33
3.2 Reinforcement Learning Agent ................................. 34
  3.2.1 Algorithm Selection ....................................... 34
  3.2.2 Feature Selection ......................................... 35
  3.2.3 Action-Values Normalization ............................... 37
  3.2.4 Implementation ........................................... 37
3.3 Search Agent .................................................... 41
  3.3.1 Algorithm Selection ....................................... 41
  3.3.2 Implementation ........................................... 42
3.4 Multiple Game Learning Agent ................................. 48

4 Evaluation ........................................................ 55
4.1 Learning in a Single Game ..................................... 55
  4.1.1 Parameter Selection ....................................... 55
    4.1.1.1 Gradient-Descent Sarsa Parameters .................. 57
    4.1.1.2 Monte Carlo Tree Search Parameters ............... 66
  4.1.2 Evaluation with the Full Set of Games ................... 71
4.2 Learning in Multiple Games .................................. 82
  4.2.1 Parameter Selection ....................................... 82
  4.2.2 Evaluation with a Partial Set of Games ................. 84

5 Related Work ...................................................... 91
5.1 Deep Q-Networks (DQN) ........................................ 91
5.2 Classical Planning .............................................. 93

6 Conclusions and Future Work .................................... 95
6.1 Conclusions ..................................................... 95
6.2 Future Work .................................................... 96

A Sarsa Training Results ............................................ 99

Bibliography ......................................................... 103
List of Figures

1.1 Screenshots of Beam Rider, Phoenix, Space Invaders and Demon Attack . . 4

2.1 The Atari 2600 console . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.2 The agent-environment interaction . . . . . . . . . . . . . . . . . . . . . . . 9
2.3 The GPI pattern using the state-value function . . . . . . . . . . . . . . . . . 14
2.4 The GPI pattern using the action-value function . . . . . . . . . . . . . . . . 14
2.5 The MCTS iteration phases . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
2.6 Example of the AMAF enhancement in Go . . . . . . . . . . . . . . . . . . . 29

3.1 Feature vector $\phi_{s,a}$ graphical representation . . . . . . . . . . . . . . . . 35
3.2 Feature vector $\phi_s$ graphical representation . . . . . . . . . . . . . . . . . 36
3.3 Example of oscillation towards reward in Ms. Pac-Man . . . . . . . . . . . . 43
3.4 Tree pruning once the best child has been selected . . . . . . . . . . . . . . . 44
3.5 Screenshots of Ms. Pac-Man and Alien . . . . . . . . . . . . . . . . . . . . . 49

4.1 Learning curves of training games with variable step-size ($\alpha$) values . . . 64
4.2 Learning curves of training games with variable exploration ($\epsilon$) values . . 65
4.3 Learning curves of training games with and without optimistic initialization 65
4.4 Screenshots of Pitfall, Montezuma’s Revenge and Freeway . . . . . . . . . . . 79
4.5 Results of Freeway and Venture when learning alone . . . . . . . . . . . . . . 80
4.6 Results of Breakout and Pong when learning alone . . . . . . . . . . . . . . . 81
4.7 Results of Atlantis, Krull and Ms. Pac-Man when learning alone . . . . . . . 82
List of Tables

2.1 Executable actions by the agent for player 1 .......................... 31
2.2 Executable actions by the agent for state and system management 32

4.1 Best combinations of the Sarsa parameters ($\alpha$, $\epsilon$ and optimistic initialization) for the training games ........................................ 59
4.2 Game ranking positions for all the combinations of $\alpha$, $\epsilon$ and optimistic initialization ......................................................... 60
4.3 Relative scores to the highest scores for the training games with $\epsilon = 0.01$ . 62
4.4 Evaluation of time and score for some combinations of the number iterations per node and the number of simulated frames in Ms. Pac-Man .... 66
4.5 Evaluation of different exploration constants $c$ in the UCT algorithm . . 67
4.6 Maximum number of AMAF visits in any node of the UCT tree for each training game ............................................................... 68
4.7 Evaluation of diverse values of the RAVE parameter in UCT-RAVE .... 69
4.8 Best results achieved with UCT with and without using RAVE .......... 71
4.9 Summary of the parameters used for evaluation in Sarsa and UCT .... 72
4.10 Scores of Sarsa and UCT compared to random and human performance . 75
4.11 Summary of the times each agent (Sarsa, UCT, random and human) has obtained the highest score in a game ................................. 75
4.12 Normalized scores for Sarsa, UCT, random and human agents ........ 76
4.13 Normalized scores for learning, random and human agents ............. 76
4.14 Scores for learning in multiple games experiments .......................... 84
4.15 Improvement index for each game combination respect to playing alone . 84
4.16 Average and highest improvements of each game when playing with another game ................................................................. 85
4.17 Scores obtained in single and multiple game learning compared to the ones scored by humans .................................................. 87
4.18 Average and highest improvements caused by each game on the other game with which it learns ................................................. 89

A.1 Results of Sarsa training for the last 1000 episodes and for all episodes (30000), along with the score obtained with the last computed parameter vector ....................................................... 101
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AI</td>
<td>Artificial Intelligence</td>
</tr>
<tr>
<td>ALE</td>
<td>Arcade Learning Environment</td>
</tr>
<tr>
<td>AMAF</td>
<td>All Moves As First</td>
</tr>
<tr>
<td>DQN</td>
<td>Deep Q-Network</td>
</tr>
<tr>
<td>FSSS</td>
<td>Forward Search Sparse Sampling</td>
</tr>
<tr>
<td>GPI</td>
<td>Generalized Policy Iteration</td>
</tr>
<tr>
<td>IW</td>
<td>Iterated Width</td>
</tr>
<tr>
<td>MCTS</td>
<td>Monte Carlo Tree Search</td>
</tr>
<tr>
<td>MDP</td>
<td>Markov Decision Process</td>
</tr>
<tr>
<td>MROM</td>
<td>Mask Read Only Memory</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RAVE</td>
<td>Rapid Action Value Estimation</td>
</tr>
<tr>
<td>ROM</td>
<td>Read Only Memory</td>
</tr>
<tr>
<td>TD</td>
<td>Temporal Difference</td>
</tr>
<tr>
<td>TIA</td>
<td>Television Interface Adapter</td>
</tr>
<tr>
<td>UCB</td>
<td>Upper Confidence Bounds</td>
</tr>
<tr>
<td>UCT</td>
<td>Upper Confidence Bounds for Trees</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Context

Artificial Intelligence (AI) is the science and engineering of making intelligent machines, especially intelligent computer programs [1]. Although the term was coined by John McCarthy in 1955 [2], the research of AI started in the 1940s.

One of the first notable contributions to AI was made by the mathematician Alan Turing. His paper *Computer Machinery and Intelligence* [3] brought up the main concern of AI: create machines such that a human cannot tell whether it is actually a human or a machine. That is, machines that perform with human-level intelligence.

Since then, hundreds of papers have been published until now with the aim of achieving human-level intelligence in multiple areas: board games, robotics, data mining, video games and conversational software among many others.

A remarkable goal of AI researchers is to formulate algorithms which apply to any kind of domain. Several families of algorithms (e.g. search, learning, evolutionary) have emerged through history to solve different kinds of problems.

Nowadays, game-playing domains have received much attention from the AI research community. The reason for this is that games are usually a good test bed for diverse methods to
since it is often difficult to deal with them because of their large state spaces (a common issue in real world problems).

The main goal in AI when used in game-playing domains is to beat humans (e.g. surpass their scores or winning more games than them). A well-known example of this purpose occurred when IBM’s super computer, Deep Blue, won the world champion Garry Kasparov at chess in 1997 [4]. A search strategy was used in this case (concretely, alpha-beta search).

Also during the 1990s, Gerald Tesauro developed a program called TD-Gammon which learned to play backgammon with big success (it achieved master level) [5]. It was created by mixing reinforcement learning with artificial neural networks. TD-Gammon made human experts realize that some of their strategies were not as good as they thought.

Go, which is the most played board game in Asia, is one of the most difficult games to deal with because of its vast state space and high branching factor (specially in the 19x19 form). Until 2006 with the appearance of the UCT (Upper Confidence Bounds for Trees) algorithm [6], there was not any method (e.g. alpha-beta) that could perform with master level in a 9x9 board [7].

Beating humans is not an easy task, but finding an optimal solution of a game that makes a program invincible is even more difficult. Some researchers have recently proposed a new algorithm called CFT$^+$ that deals with this problem in poker (concretely, in the variant Heads-Up Limit Hold’Em) [8].

The checkers game has also been a focus of interest since the 1950s. During the 1990s an AI program called Chinook [9] was proclaimed the champion of the man-machine checkers contest. A decade later, in 2007, it was solved [10].

Apart from the mentioned board games, there are many others with research interest such as Othello, bridge or Magic the Gathering.

However, not only board games are currently considered, but also video games. Some video games are even more problematic in terms of computational complexity than board
games because each frame of the game could be interpreted as a different state\(^1\); therefore, if a game runs at 60 frames/second, then the state space becomes enormous as time passes.

Some examples of video games in which AI has been implemented are: Tetris\([11]\), Angry Birds\(^2\), Nintendo games (e.g. Super Mario Bros)\([12]\) or Atari games (e.g. Ms. Pac-Man)\([13]\).

1.2 Problem

As explained in section 1.1, developing agents for video games with the intention to beat humans is an important milestone in AI. Since these tasks are complex to solve, they become an important area to explore in the search of techniques that might work well in other fields.

In this thesis, the chosen domain to perform such tests is formed by the games of the Atari 2600 system. Hundreds of games were programmed for this video game console (some of them were ports of other systems), such as the widely known Space Invaders, Ms. Pac-Man, Freeway, Breakout or Beam Rider. The goal in this group of games is to obtain the highest possible score (in the best case, a higher score than the one obtained by humans).

The main approach used in this thesis is reinforcement learning. Algorithms of this kind, as the name states, learn how to play a game after playing it several times (e.g. play Space Invaders 30000 times).

The learning process is based on applying actions to the game which results in an state change. Action selection is done by taking into account numerical signals called rewards, which are obtained together with the state after applying an action. These rewards can be understood as an evaluation of the current state of the game. The purpose of reinforcement

\(^1\)To simplify the problem, it can be considered that the transition between states is done every 10 frames, for example.

\(^2\)There is a competition of building the best Angry Birds agent called ALBirds: https://aibirds.org/
learning algorithms is to maximize the sum of all these rewards (the same that a human would try to do in Atari games).

An specific challenge to face with this kind of algorithms consists in analyzing **how well learning generalizes to several games** played simultaneously. Humans are good at generalizing information between games. For example, it is easy to appreciate visually and after playing few seconds that Space Invaders, Phoenix, Demon Attack and Beam Rider are very similar games (see figure 1.1).

An alternative approach to reinforcement learning is building a **search tree** of the different states that are found during the game. Examining the entire tree of states can be very expensive in terms of memory, so methods which expand those most promising parts are the main candidates to be used in this domain.

Given the complexity of this kind of tasks, it is critical to choose the appropriate methods to treat the information given by the game. Moreover, it is also important to set a process for getting the parameters that are useful to obtain the highest possible scores.

The main tool used to do all these tasks is a framework called **ALE** (Arcade Learning Environment) [13], which was developed by the University of Alberta (Canada). This framework provides an interface that allows computer programs to communicate with Atari games, which are played thanks to the Stella emulator. Concretely, the program sends an action through the interface, and the game returns its new current state and a reward for having reached it.

To sum up, the goal of this project is to evaluate learning and search methods in the Atari 2600 games, putting special emphasis in how learning methods can generalize to several games by playing them simultaneously.
1.3 Thesis Organization

In chapter 2 some background on the topic being covered in this thesis is presented, mainly focusing on the features of the Atari 2600 and the characteristics of the methods that will be used to play the games (reinforcement learning and search methods). Moreover, the tool that is used to communicate with the Atari games, the Arcade Learning Environment (ALE), is also explained.

Chapter 3 explains the methodology used to select the programming language as well as the specific algorithms of each kind (reinforcement learning and search). The used approach to deal with learning in multiple games is also covered here.

The evaluation of the algorithms is done in chapter 4. Firstly, calibrations of the diverse parameters in each algorithm is done by using a small set of games. Then, once the parameters are set, the methods are run with the full set of games supported by ALE. The scores of these algorithms are compared to the ones obtained by a human and a random agent to evaluate how good they are. The evaluation of the process of learning in multiple games is analyzed in this chapter as well.

Other approaches that have emerged recently and have obtained successful results are briefly explained in chapter 5. Finally, conclusions and future work in this field are detailed in chapter 6.
Chapter 2

Background Theory

2.1 Atari 2600

The Atari VCS (Video Computer System), better known as Atari 2600, was a video game platform created by Atari in 1977. A summary of the main technical features of the system are explained in the following paragraphs [14].

The processor of the console was an 8-bit MOS Technology 6507, a cheaper version of the 8-bit MOS Technology 6502. The number of lines of memory, whose function is to determine which byte is read or written, was the key feature that differentiated both processors. While the 6502 had 16 lines of memory (64K of addressable memory), the 6507 had 13 lines of memory (8K of addressable memory). The frequency of this component was 1.19 MHz.

Atari 2600 games consisted of cartridges with a ROM (Read Only Memory). Concretely, the ROM contained in the games was MROM (Mask ROM), a kind of ROM which is hardwired when manufactured and is neither reprogrammable nor reflashable. All the code and data of the game were saved in the ROM.

The memory addressed by the processor was the ROM memory of the game cartridge and, as said before, the processor could address up to 8K of memory. However, at that time 8K
Background Theory

of MROM was quite expensive, so it was decided to restrict addressable memory to 4K. Years later, cartridges with 8K MROM were allowed to be used using a technique called bank switching.

The **RAM (Random Access Memory)** was the entire store for the Atari 2600. It consisted of 128 bytes containing the code variables, the game state, the score and the stack (i.e. program state). Unlike contemporary systems, Atari games ran directly from the ROM and not from the RAM; that is, the RAM did not hold the program to be executed. Some cartridges included up to 256 bytes of extra RAM.

A chip called **TIA (Television Interface Adapter)** enabled the sound and image functions in the Atari console. The image was shown through 192 horizontal scan lines; each line had to be displayed individually since the image was not stored in the RAM of the console. Therefore, one of the biggest issues when programming games consisted in synchronizing the processor instructions with the television via the TIA. Besides, the TIA could generate two sound channels in which the tone, the volume and the type of sound could be controlled.

The console had some switches to turn it on/off, reset it, change the difficulties, change the game and change from color to gray scale and vice versa. Furthermore, there were ports for diverse kinds of controllers: joystick, paddle, keypad or trackball among others.

![Figure 2.1: The Atari 2600 console](http://goo.gl/84MTjg)

---

1Image extracted from [http://goo.gl/84MTjg](http://goo.gl/84MTjg)
2.2 Sequential Decision Making

Artificial Intelligence problems are usually represented through the interaction between two elements: the agent and the environment. The agent makes decisions according to information received from the environment; in contrast, the environment reacts to those decisions and sends information about itself to the agent [15].

The interaction between the agent and the environment is clearly cyclic: the agent feeds the environment with information and vice versa. This interaction takes place at discrete time steps $t = 0, 1, 2, 3, \ldots$. The following data is exchanged at each time step $t$ (see figure 2.2):

1. The agent receives a representation of the environment $s_t \in S$ called state, where $S$ is the set of all possible states.

2. The agent selects an action $a_t \in A(s_t)$, where $A(s_t)$ is the set of applicable actions to state $s_t$.

3. As a consequence of action $a_t$, the environment answers with a representation of its current state $s_{t+1} \in S$ and a numerical signal called reward $r_{t+1} \in \mathbb{R}$ for being at that state.

![Figure 2.2: The agent-environment interaction [15]](image)

The agent selects an action depending on a policy $\pi$, which consists in a mapping from a state $s$ to an action $a$. There are two kinds of policies:

- **Deterministic:** The same action $a$ is selected in state $s$. 
\[ \pi(s) = a \]

- **Non-Deterministic:** The selection of an action \( a \) in state \( s \) is subject to a probability distribution.

\[ \pi(s, a) = P(a|s) \]

There are two types of interactions: episodic and continuing. **Episodic** interactions, as the name states, are divided into subsequences called episodes; all episodes finish with a terminal state. An example of episodic interaction takes place in Space Invaders: each episode starts with the player having three lives and finishes when the player has been killed three times. On the other hand, **continuing** interactions are not divided into episodes, so there is not any terminal state (i.e. the final time step \( T \) tends to infinity).

The **goal** of the agent is to maximize the amount of reward it receives. To mathematically treat this intention, the **return** at time \( t \), \( R_t \), is presented as the function of future rewards which must be maximized.

To generalize to the episodic and the continuing interaction, a **discount factor** \( \gamma \in [0, 1] \) is introduced in the return formulation. Moreover, it allows to give more or less weight to future rewards; therefore, return can be computed taking the future rewards into account \((\gamma = 1)\) or not \((\gamma = 0)\).

\[ R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{T} r_{t+k+1}\gamma^k \]

The main difficulty in this problem is having a good policy that allows us to choose the actions which will eventually give the highest possible return. In the following chapters different kinds of policies will be explained; in general, most of them are based on comparing numerical values of actions, each representing the desirability of applying an action in the current state. Those actions with the highest value are called **greedy** actions, whereas the other ones are labeled as **non-greedy** actions.
To maximize the return, it is important to balance between exploitation (greedy action selection) and exploration (non-greedy action selection) since it is important not only to take advantage of actions with which we get huge rewards, but also to use actions which might give highest rewards in the long-term (delayed rewards).

2.3 Markov Decision Processes

An state is said to have the Markov property if, and only if, the following equality holds [15]:

$$P(s_{t+1} = s', r_{t+1} = r|s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0) = P(s_{t+1} = s', r_{t+1} = r|s_t)$$

That is, the response of the environment at time $t + 1$ only depends on the state of the previous time step $t$ instead of the full history of states, actions and rewards. Therefore, it can be said that $s_t$ already includes information about what happened in previous time steps.

A Markov Decision Process (MDP) is a model of stochastic sequential decision problems in fully observable environments. All states in the environment hold the Markov property [4, 15]. Formally, it is a tuple $\langle S, A, P^a_{ss'}, R^a_{ss'}, \gamma \rangle$ where the elements are:

- A set of states $S$.
- A set of actions $A$.
- A matrix of transition probabilities between states $P^a_{ss'} = P(s_{t+1} = s'|s_t = s, a_t = a)$.
- A reward function $R^a_{ss'} = E(r_{t+1}|s_t = s, a_t = a, s_{t+1} = s')$.
- A discount factor $\gamma \in [0, 1]$.

If $S$ and $A$ are finite, then the MDP is called finite MDP.
As previously mentioned, the return at time \( t, R_t \), is the sum of future rewards starting from time \( t \). **Value functions** use the concept of expected return in order to give an estimation of the desirability of being in an state (state-value function) or of applying an action in a specific state (action-value function). These functions are defined respect to particular policies [15].

The **state-value function** \( V^\pi(s) \) of a MDP is the expected return starting from state \( s \) and then following policy \( \pi \):

\[
V^\pi(s) = E_\pi \{ R_t | s_t = s \} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s \right\}
\]

The **action-value function** \( Q^\pi(s,a) \) of a MDP is the expected return starting from \( s \), selecting \( a \) and then following policy \( \pi \):

\[
Q^\pi(s,a) = E_\pi \{ R_t | s_t = s, a_t = a \} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a \right\}
\]

Value functions can be reformulated in order to write them recursively; that is, write a value of a state depending on the value of another. The Bellman equation for \( V^\pi(s) \), which is derived from the previous definition of the state-value function, follows the mentioned recursive formulation [15]:

\[
V^\pi(s) = \sum_{a \in \mathcal{A}(s)} \pi(s,a) \sum_{s' \in \mathcal{S}} P_{ss'}^a [R_{ss'}^a + \gamma V^\pi(s')]
\]

The goal is to find the optimal policy which gives us the optimal value function; that is, the policy that tells which actions lead us to the maximum possible return. As before, there are two value functions: the optimal state-value function and the optimal action-value function:

\[
V^*(s) = \max_\pi V^\pi(s), \forall s \in \mathcal{S}
\]

\[
Q^*(s,a) = \max_\pi Q^\pi(s,a), \forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)
\]
The Bellman optimality equation for $V^*$ tells that the value of a state following an optimal policy equals the return of the best eligible action from that state:

$$V^*(s) = \max_{a \in A(s)} E \left\{ r_{t+1} + \gamma \max_a V^*(s_{t+1}) | s_t = s, a_t = a \right\}$$

$$= \max_{a \in A(s)} \sum_{s' \in S} P^{a}_{ss'} \left[ R^{a}_{ss'} + \gamma V^*(s') \right]$$

In the case of the last equation, there is only one solution which can be found by solving a system of $N$ equations with $N$ unknowns, where $N$ is the number of states. However, transition probabilities $P^{a}_{ss'}$ and expected rewards $R^{a}_{ss'}$ must be known in order to get the solution. Besides, this equation is not linear, so it is much more difficult to solve than the Bellman equation with a fixed policy.

On the other hand, the Bellman optimality equation for $Q^*$ is:

$$Q^*(s, a) = E \left\{ r_{t+1} + \max_{a'} Q^*(s_{t+1}, a') | s_t = s, a_t = a \right\}$$

$$= \sum_{s'} P^{a}_{ss'} \left[ R^{a}_{ss'} + \gamma \max_{a'} Q^*(s', a') \right]$$

### 2.4 Reinforcement Learning

Reinforcement Learning is a set of algorithms whose purpose is to learn what to do in a given situation in order to maximize a numerical reward signal [15]. The learner will have to deal with the exploration-exploitation problem (see section 2.2) since it will have to discover which actions give more reward in the future (delayed rewards).

#### 2.4.1 The Generalized Policy Iteration Pattern

In general, reinforcement learning methods follow a pattern of policy evaluation and improvement called GPI (Generalized Policy Iteration). The purpose is to converge to the optimal policy $\pi^*$ and to the optimal value function $V^*$.
The **evaluation process** consists in computing the value function $V$ according to the current policy $\pi$ (that is, compute $V^\pi$).

On the other hand, the **improvement process** consists in changing the policy $\pi$ such that if we used it, then $V$ would be obtained. In other words, $\pi$ becomes greedy with respect to $V$.

In figure 2.3, it is shown how these two processes interact with each other. The loop ends when stability is reached; that is, neither the policy nor the value function are changing since they are the optimal ones.

![Figure 2.3: The GPI pattern using the state-value function [15]](image)

The policy improvement process, however, does not usually use the state-value function. Instead, the action-value function is employed because it allows to get the policy more easily (it is just a matter of convenience).

![Figure 2.4: The GPI pattern using the action-value function [15]](image)
One of the most used methods for policy improvement is the \textbf{\(\epsilon\)-greedy policy}. It consists in choosing a random action with probability \(\epsilon\) or the greedy action (the one with the highest action-value) with probability \(1 - \epsilon\). Therefore, the following probability distribution is followed:

\[
\pi(s, a) = \begin{cases} 
1 - \frac{\epsilon}{|A(s)|} & \text{if } a = \text{arg max}_{a \in A(s)} Q(s, a) \\
\epsilon & \text{otherwise}
\end{cases}
\]

where \(|A(s)|\) is the number of executable actions on state \(s\). In case there are several state-action pairs with the same \(Q(s, a)\) value, then one of them is randomly selected.

There are two kinds of learning methods:

- \textbf{On-Policy}: The policy \(\pi\) is used for evaluation and improvement. Examples: Sarsa, Actor-Critic methods.

- \textbf{Off-Policy}: The policy \(\pi\) improved is different from the policy \(\pi'\) used for evaluation. Examples: Q-Learning, R-Learning.

\subsection{2.4.2 Function Approximation}

In reinforcement learning problems, value functions are usually represented as tables: for each possible state, its value is saved. This approach is prohibitively expensive in terms of memory if there are many states (to the extent that there is not enough memory to save them).

To address this problem, a function that generalizes to all states is used. That is, having an state as input, the function returns its approximated value. This method is called function approximation.

By using function approximation, the reinforcement learning problem turns into a supervised learning problem: some training examples are used to create a model which will be used to compute the value of other examples (in our case, the examples are states). There exist many techniques for treating supervised learning problems such as artificial neural networks or decision trees.
The value of a state may change between time steps; therefore, the value of a state $s$ at time $t$ is denoted by $V_t(s)$. In function approximation, each value of a state at time $t$ is expressed depending on a parameter vector $\theta_t$. This parameter vector must have less components than the number of possible states; if they had the same number, we would face the same memory issues as before. It is important to note that if a parameter is changed, the value of several states might be altered.

The main goal in function approximation consists in finding the optimal vector of parameters $\theta^*$. If this optimal vector is found, then it is said that we have found $V^*$, i.e. the optimal value for all states of the domain.

2.4.2.1 Gradient-Descent Methods

Gradient-Descent methods are widely used to perform function approximation in reinforcement learning [15].

As explained in the introduction of section 2.4.2, there is a parameter vector $\theta_t$ with a fixed number of components. Besides, there is also a differentiable function of $\theta_t$, $V_t(s) \forall s \in S$.

In these methods, the parameter vector adjusts the value of its components in the direction of highest decrease of the error by a small amount:

$$\theta_{t+1} = \theta_t + \alpha [V^\pi(s_t) - V_t(s_t)] \nabla_{\theta_t} V_t(s_t)$$

where $\alpha > 0$ is the step-size parameter and $\nabla_{\theta_t} V_t(s_t)$ (gradient of $V_t(s_t)$ respect to $\theta_t$) the direction in which the error decreases faster. Nevertheless, the core of the equation is $V^\pi(s_t) - V_t(s_t)$, which represents the error of the current estimation $V_t(s_t)$ respect to the real value of $s_t$ for policy $\pi$, $V^\pi$.

The main problem with the shown update is that we usually do not have the $V^\pi(s_t)$ values, so we need an estimation. The estimated value of $V^\pi(s_t)$ is obtained through
bootstrapping (that is, using the value of another state):

\[ V_\pi(s_t) \approx r_{t+1} + \gamma V_t(s_{t+1}) \]

where \( \gamma \) is the discount factor, and \( r_{t+1} \) the reward for being at state \( s_{t+1} \).

The update rule can be rewritten in the following way [15]:

\[
\theta_{t+1} = \theta_t + \alpha \delta_t \nabla_\theta V_t(s_t) \tag{2.1}
\]

\[
\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t) \tag{2.2}
\]

Equations 2.1 and 2.2 can be rewritten to be used with action-values \( Q_t(s_t, a_t) \):

\[
\theta_{t+1} = \theta_t + \alpha \delta_t \nabla_\theta Q_t(s_t, a_t) \tag{2.3}
\]

\[
\delta_t = r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t) \tag{2.4}
\]

There are two kinds of function approximation: linear (explained in 2.4.2.2) and non-linear, which usually consists in a multi-layer artificial neural network where backpropagation is used to compute gradients (for example, Tesauro’s TD-Gammon [5]).

A common problem arises when using non-linear function approximation to represent the action-values: the learning process tends to be unstable or to diverge. However, there exist some strategies to deal with these problems (read [16] for more information).

Furthermore, as described in [15], off-policy bootstrapping methods (such as Q-Learning) can diverge even for linear function approximation.

2.4.2.2 Linear Function Approximation

Linear function approximation is a gradient-descent function approximation method in which apart from the vector of parameters \( \theta_t \), there is also a vector of features in state \( s \),
Background Theory

\( \phi_s \), with the same number of components as \( \theta_t \).

In this case, \( V_t(s) \) is computed as the dot product between \( \theta_t \) and \( \phi_s \):

\[
V_t(s) = \sum_{i=1}^{n} \theta_t(i) \phi_s(i)
\]

where \( n \) is the number of components of both vectors.

Moreover, the gradient of the state-value function \( V_t(s_t) \) with respect to \( \theta_t \) is the feature vector:

\[
\nabla_{\theta_t} V_t(s_t) = \phi_s
\]

The previous equations apply to action-values, \( Q_t(s_t, a_t) \), as well:

\[
Q_t(s, a) = \sum_{i=1}^{n} \theta_t(i) \phi_{s,a}(i)
\]

\[
\nabla_{\theta_t} Q_t(s_t, a_t) = \phi_{s,a}
\]

A single optimum \( \theta^* \) exists in linear methods. Therefore, if the method is guaranteed to converge, then it will to a global optimum.

There is a particular kind of features called binary features which, as its name states, can have two values: 0 or 1. In this case, the previous dot product would be equivalent to sum the parameters corresponding to non-zero positions in the feature vector.

2.4.3 Sarsa

Sarsa is an on-policy learning algorithm, that is, the same policy \( \pi \) is used for evaluation and improvement (see section 2.4.1). The action-values, \( Q(s, a) \), are updated in the following way:

\[
Q(s_t, a_t) = Q(s_t, a_t) + \alpha \left[ r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]
\]
where $\alpha$ is the learning rate and $\gamma$ is the discount factor. This algorithm bootstraps: action-values are updated taking into account action-values of other state-action pairs.

The use of the tuple $< s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1} >$ gave the name to this algorithm. The updating process followed in the tabular form of Sarsa is shown in algorithm 1.

**Algorithm 1** Tabular Sarsa algorithm [15]

1: Initialize $Q(s,a)$ arbitrarily  
2: for each episode do  
3:    Initialize $s$  
4:    repeat for each step of the episode  
5:        Take action $a$, observe $r, s'$  
6:        Choose $a'$ from $s'$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)  
7:        $Q(s,a) \leftarrow Q(s,a) + \alpha [r + \gamma Q(s',a') - Q(s,a)]$  
8:    until $s$ is terminal  
9: end for

An important property of tabular Sarsa is that it is guaranteed to converge to an optimal policy and action-value function if all the states are visited an infinite number of times.

However, as described in section 2.4.2, in many cases it is not possible to use a table to represent a value function; it is necessary to use function approximation instead. The variant of Sarsa for function approximation is known as gradient-descent Sarsa. In the next paragraphs this kind of Sarsa algorithm is explained in the context of linear function approximation (where it is mostly used).

The update of the vector of parameters in gradient-descent Sarsa with linear function approximation is done in the same way as indicated in section 2.4.2.1 for action-values:

$$\theta_{t+1} = \theta_t + \alpha \delta_t \phi_{s,a}$$

$$\delta_t = r_{t+1} + \gamma Q_t(s_{t+1},a_{t+1}) - Q_t(s_t,a_t)$$

Algorithm 2 shows all the steps performed in gradient-descent Sarsa with linear function approximation.
Algorithm 2 Gradient-Descent Sarsa Algorithm with Linear Function Approximation [15]

1: Initialize $\theta$ arbitrarily
2: for each episode do
3: $s, a \leftarrow$ initial state and action of episode
4: $\phi_{s,a} \leftarrow$ set of features present in $s, a$
5: repeat for each step of the episode
6: Take action $a$, observe reward $r$, and next state $s'$
7: $\delta \leftarrow r - \sum_i \theta(i)\phi_{s,a}(i)$
8: with probability $1 - \epsilon$
9: for all $b \in \mathcal{A}(s)$ do
10: $\phi_{s',b} \leftarrow$ set of features present in $s', b$
11: $Q_b \leftarrow \sum_i \theta(i)\phi_{s',b}(i)$
12: end for
13: $a' \leftarrow \arg\max_a Q_a$
14: else
15: $a' \leftarrow$ a random action $\in \mathcal{A}(s')$
16: $\phi_{s',a'} \leftarrow$ set of features present in $s', a'$
17: $Q_{a'} \leftarrow \sum_i \theta(i)\phi_{s',a'}(i)$
18: end with
19: $\delta \leftarrow \delta + \gamma Q_{a'}$
20: $\theta \leftarrow \theta + \alpha \delta \phi_{s,a}$
21: $s \leftarrow s'$; $a \leftarrow a'$; $\phi_{s,a} \leftarrow \phi_{s',a'}$
22: until $s$ is terminal
23: end for

On-policy bootstrapping methods, like Sarsa, with linear gradient-descent function approximation converge to a solution with mean-squared error bounded by $\frac{1}{1 - \gamma}$ times the minimum possible error [15].

Sarsa has proved to be a good method in contexts like the RoboCup contest [17] and in some 2D games such as Grid World or Snake [18].

2.4.4 Optimistic Initialization

Optimistic initialization is a method often used in reinforcement learning to encourage exploration [15].

The common approach consists in initializing the action-values for all actions with a high value (depending on the domain of application). Therefore, any action chosen by the
agent will be greedy since the reward obtained will not be higher than its initial value; consequently, all actions will be tested in the initial phases of the agent-environment interaction.

The drawback of the previous approach is that it is domain-dependent: the reward scale must be known in order to set a reference action-value to encourage exploration.

A domain-independent approach is explained at [19]; so it does not require any previous knowledge of the domain of application. The main idea is to normalize all rewards $r_t$ by the absolute value of the first non-zero reward $|r_{1st}|$. Besides, the normalized reward is also shifted by $\gamma - 1$:

$$\tilde{r}_t = \frac{r_t}{|r_{1st}|} + (\gamma - 1)$$

where $\gamma$ is the discount factor.

In this domain-independent approach, there is also provided a termination reward $r_{end}$. It is needed because the previous shift causes the agents to try to finish the episodes as fast as possible to avoid negative rewards. This termination reward is subtracted from the previous normalized reward at the end of the episode (in episodic tasks):

$$\tilde{r}_t = \frac{r_t}{|r_{1st}|} + (\gamma - 1) - r_{end}$$

$$r_{end} = \gamma^{T-k+1} - 1$$

where $k$ is the number of steps in the episode and $T$ is the maximum number of steps.

2.5 Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search (MCTS) is a family of algorithms which, as its name states, are based in the construction of search trees. These trees are built in an incremental and partial way since they only expand those areas which seem to be promising (i.e. the ones which are likely to lead to a higher return).
Each node in the tree has a state, the number of times it has been visited, and a numerical value which is approximated by performing random simulations. Actually, this value is the expected accumulated reward to be obtained at the end of the game, and they will be used to build the tree following a best-first strategy [20].

This section explains the main principles of the MCTS methods and their main features. Besides, the UCT (Upper Confidence Bounds for Trees), which is the most used MCTS method today, will be shown along with possible enhancements of the algorithm.

### 2.5.1 General Approach

The general MCTS algorithm is based on the repetition of several phases until a previously set computational budget (e.g. memory, time or number of iterations) is exceeded. The phases involved in this loop are the following:

1. **Selection**: Consists in traversing the tree looking for a node to expand. An expandable node is a node whose state is not terminal and over which an applicable action has not been used yet. Until an expandable node is found, the selection of the next node in the tree is done according to an assigned numerical value (the expected accumulated reward starting from that state).

2. **Expansion**: An available action (i.e. an action that has not been used yet) is applied over the state of the selected node. A new child node is added to the expanded node with the resulting state of the action.

3. **Simulation**: From the last created node, a simulation is run with the objective of getting a value estimation for it. The actions executed in this stage are usually randomly selected.

4. **Backpropagation**: The value of the new node is propagated up to the root of the tree. Each node in the path will update its information (e.g. number of visits and expected accumulated reward).
These stages (except for the backpropagation) are grouped in two different policies: the **tree policy** (selection and expansion) and the **default policy** (simulation). A graphical summarization is shown in figure 2.5.

![Figure 2.5: The MCTS iteration phases [20]](image)

Once the loop condition no longer holds, the action of the **best child** of the root node is selected; that is, the action which was applied on the root state to get the best child state. There are diverse criteria to define the which child is the best [21]:

- **Max child**: The child with the highest reward.
- **Robust child**: The most visited child.
- **Max-Robust child**: The child with highest number of visits and the highest reward. If there is not any max-robust child, then it is better to keep iterating until one appears.
- **Secure child**: The child which maximizes a lower confidence bound.

All these steps are summarized in algorithm 3. The notation is the following: $v_0$ is the root node whose state is $s_0$, $v_l$ is the selected child of an expandable node and $\Delta$ is the value of the selected node state $s(v_l)$. Finally, $a(\text{BESTCHILD}(v_0))$ stands for the action that links the root node with the best child node.
Algorithm 3 General MCTS approach [20]

1: function MctsSearch($s_0$)  
2: create root node $v_0$ with state $s_0$  
3: while within computational budget do  
4: \hspace{1em} $v_l \leftarrow$ TreePolicy($v_0$)  
5: \hspace{1em} $\Delta \leftarrow$ DefaultPolicy($s(v_l)$)  
6: \hspace{1em} Backup($v_l, \Delta$)  
7: end while  
8: return $a(\text{BestChild}(v_0))$  
9: end function

2.5.2 Characteristics

The main characteristics of MCTS methods are the following [20]:

- **Aheuristic**: They are domain-independent; therefore, they can be applied in any problem modeled as a tree. Nevertheless, improvements can be implemented by using knowledge of the domain.

- **Anytime**: The values of the nodes are always updated since the value $\Delta$ obtained from the simulation is propagated immediately. Therefore, the best action eligible from the root can be returned in any moment without doing more computations.

- **Asymmetric**: The tree is generated to select the most promising actions, so there are areas where the density of nodes will be much higher than in others.

The mentioned features are some of the main advantages of MCTS. However, there is an important drawback which is common in tree-based methods: if the branching factor and the depth of the tree are high, then the time and memory required to build the tree increase. Moreover, random simulations might not be appropriate in environments where the reward is difficult to get.
2.5.3 The UCT Algorithm

The UCT (Upper Confidence Bounds for Trees) is the most popular MCTS method nowadays. Kocsis and Szervespári formulated UCT in 2006 as a MCTS in which the selection mechanism (i.e. tree policy) had low error probability if it stopped early and that converged given sufficient time and memory [6].

A way to fulfill the previous requirements was to balance exploration and exploitation (a common issue in AI problems). The reason behind this fact is that if different actions are tested several times, then it is easier to tell which actions are the best for a given situation with lower probability of error.

The policy chosen to balance exploitation and exploration was UCB (Upper Confidence Bounds) [22], which had been used in the bandits problem [15]. Concretely, it was UCB1 the one used by Kocsis and Szervespári as the tree policy in UCT. By using this policy, the selection of the next node in the tree is represented as a multi-armed bandit problem. Furthermore, it was shown that UCT converges to the optimal solution given enough time and memory [23] unlike other Monte Carlo methods.

Nevertheless, UCT can take exponential time to find the best action \( a \) from a state \( s \). Other MCTS methods, such as FSSS (Forward Search Sparse Sampling), have better guarantees in terms of time than UCT [24]. However, most researchers keep using UCT instead of other MCTS since it usually provides better results.

In UCT, the next child node \( j \) in the traversal (selection phase) is selected to maximize:

\[
X_j + 2C_p \sqrt{\frac{2\ln(n)}{n_j}}
\]

where \( X_j \) is the average value for the child node \( j \), \( C_p > 0 \) is a constant, \( n \) is the number of times that the parent node has been visited and \( n_j \) the number of times that the child node \( j \) has been visited. In case there are several child nodes with the maximum value, then one of them is randomly selected.
As it can be seen, there are two parts in the formula separated by the sum:

- **Exploitation term:** Since $X_j$ is the average value for the child node $j$, those nodes with highest values will tend to be selected more often.

- **Exploration term:** Represents how often the child node $j$ has been visited in relation to its parent. The less times the child node is visited compared to its parent, the higher value this term will have; in fact, if $n_j = 0$ the value of the node to be selected will be $\infty$, so all child nodes can be eventually selected. Therefore, until all child nodes have been selected once, the exploitation term is not important.

Algorithm 4 represents all the functions of UCT as explained in this section. The function UctSearch($s_0$) is almost the same as MctsSearch($s_0$) in algorithm 3. However, in the UCT case, the presence of variables for counting the visits ($N(v)$) and the score ($Q(v)$) of the node is emphasized. Also note that the UCB constant at BestChild($v$, $c$) is set to 0 when the loop has finished so as to select the greedy action (in case of tie, a random action would be selected among the greedy ones).
Algorithm 4 UCT [20]

1: function UctSearch(s₀)
2:     create root node v₀ with state s₀
3:     while within computational budget do
4:         v₁ ← TreePolicy(v₀)
5:         Δ ← DefaultPolicy(s(v₁))
6:         Backup(v₁, Δ)
7:     end while
8:     return a(BestChild(v₀, 0))
9: end function

10: function TreePolicy(v)
11:     while v is nonterminal do
12:         if v not fully expanded then
13:             return Expand(v)
14:         else
15:             v ← BestChild(v, C_p)
16:         end if
17:     end while
18:     return v
19: end function

20: function BestChild(v, c)
21:     return arg max_{v' ∈ children of v} \frac{Q(v')}{N(v')} + c√\frac{2 ln N(v)}{N(v')}
22: end function

23: function Expand(v)
24:     choose a ∈ untried actions from \mathcal{A}(s(v))
25:     add a new child v' to v with s(v') = f(s(v), a) and a(v') = a
26:     return v'
27: end function
Algorithm 4 UCT (continued) [20]

28: function DefaultPolicy(s)
29:     while s is non-terminal do
30:         choose \( a \in \mathcal{A}(s) \) uniformly at random
31:         \( s \leftarrow f(s, a) \)
32:     end while
33:     return reward for state \( s \)
34: end function

35: function Backup(\( v, \Delta \))
36:     while \( v \) is not null do
37:         \( N(v) \leftarrow N(v) + 1 \)
38:         \( Q(v) \leftarrow Q(v) + \Delta(v, p) \)
39:         \( v \leftarrow \text{parent of } v \)
40:     end while
41: end function

2.5.4 All Moves As First (AMAF)

All Moves As First (AMAF) is a tree policy enhancement of the MCTS method; that is, the selection criteria is different from the one previously described. However, to apply AMAF it is necessary to change the way in which updating is done during the backpropagation stage.

The main idea of AMAF consists in changing the way in which node updates are done. Two kinds of updates are differentiated when UCT and AMAF are combined [25]:

- **Standard update**: Only the nodes traversed during selection are updated in the backpropagation phase.

- **AMAF update**: The nodes traversed during selection and some of their siblings are updated in the backpropagation phase.
The AMAF update, as explained before, is not performed in all the siblings but only in those whose state has been traversed during the simulation phase. Then, it is essential to save the history of all the traversed states during the simulation.

An example of AMAF update is shown in figure 2.6 with the board game Go. In this case, black has played C2 and white has played A1. Then, the simulation step starts and the following sequence of states is traversed until black wins (terminal state): Black B1, White A3 and Black C3. After that, the backpropagation step starts, so the sibling nodes of Black C2 and White A1 which match the states of the simulation must also update their AMAF statistics: White A3 (sibling of White A1), Black B1 and Black C3 (siblings of Black C2). The updated siblings are marked with an asterisk (*).

![Figure 2.6: Example of the AMAF enhancement in Go [25]](image)

In \(\alpha\)-AMAF two sets of counts (for visits and reward) are saved for each node: one for each kind of update (standard and AMAF). An \(\alpha\) parameter is used to determine the value which will be used during the tree policy:

\[
\alpha \cdot s_{AMAF} + (1 - \alpha) \cdot s_{std}
\]

where \(s_{AMAF}\) is the score obtained with AMAF updates, and \(s_{std}\) is the score obtained with standard MCTS updates.

RAVE (Rapid Action Value Estimation) is a variant of \(\alpha\)-AMAF which has proven to be a solid add-on for UCT in Computer Go [7]. In this case the \(\alpha\) parameter decreases as the
number of AMAF visits increases in each node:

\[ \alpha = \max \left( \frac{\text{parameterVal} - v_{AMAF}}{\text{parameterVal}}, 0 \right) \]

where parameterVal is the number of AMAF playouts \( v_{AMAF} \) from which \( \alpha \) will start to be 0.

### 2.6 Arcade Learning Environment (ALE)

The Arcade Learning Environment (ALE)\(^2\) is a framework which provides an interface for developing general AI agents for Atari 2600 games [13]. At the time of writing this thesis, the current version of ALE is 0.4.4 and 61 games are supported.

The code of ALE clearly divides the two main parts in an AI system (see section 2.2): the agent and the environment. The agent implementation depends on the person using the framework, whereas the environment is already given with an interface to interact with it. This environment interface is directly connected to Stella\(^3\), an emulator that implements Atari 2600 behavior in software.

Agents can be written in C++ language since the framework is coded in C++; besides, agents can be also programmed using other languages (e.g. Java, Python) through the use of pipes. If C++ is used as the programming language, it is possible to add more supported games by inheriting the abstract class used to represent a game. However, it is necessary to know in which memory addresses are the score and the lives saved among other knowledge required by the interface.

The interaction with Atari games is episodic (see section 2.2). The measure to know the progress of the interaction is the frame, so some of the configuration parameters of ALE are related to it. For example, a way to finish an episode if a terminal state has not been found yet, is to set a maximum number of frames for the episode. In general, each time

\(^2\)ALE website: [http://www.arcadelearningenvironment.org/](http://www.arcadelearningenvironment.org/)
an action is executed for one single frame; however, it is possible to change the number of frames during which the action is applied (also called frame skip) in the configuration.

At each step of the interaction between the agent and the environment, the agent can get diverse information from the environment:

- **RAM**: An array of the 128 RAM bytes present in the Atari 2600.

- **Reward**: The reward obtained for the current state of the environment.

- **Screen**: An array of bytes that represents the pixel values of the display.

- **State**: It contains the current frame number since the beginning of the current episode and since the beginning of the first episode. In addition, it also saves the values for the right and left paddles.

  The state is actually saved in form of a string which can be reloaded into the environment (an essential feature for algorithms like UCT).

Moreover, the agent can apply 18 actions for each player (table 2.1). For player 1, actions are the ones from 0 to 17; for player 2, actions go from 18 to 35.

<table>
<thead>
<tr>
<th>Action</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>noop</td>
<td>0</td>
<td>fire</td>
<td>up</td>
</tr>
<tr>
<td>right</td>
<td>3</td>
<td>left</td>
<td>down</td>
</tr>
<tr>
<td>up-right</td>
<td>6</td>
<td>up-left</td>
<td>down-right</td>
</tr>
<tr>
<td>down-left</td>
<td>9</td>
<td>up-fire</td>
<td>right-fire</td>
</tr>
<tr>
<td>left-fire</td>
<td>12</td>
<td>down-fire</td>
<td>up-right-fire</td>
</tr>
<tr>
<td>up-left-fire</td>
<td>15</td>
<td>down-right-fire</td>
<td>down-left-fire</td>
</tr>
</tbody>
</table>

**Table 2.1**: Executable actions by the agent for player 1

There are also three additional actions related to state and system management (table 2.2). The save-state action puts the current environment state into the environment’s stack of states, while the load-state does the opposite: get the state of the top of the stack and pops it. The system-reset action resets the environment.
The entire set of actions is not always used by all games, most of them have a subset of actions which have an impact in the game. All games have a method for obtaining this subset of actions.

Another important functionality that ALE incorporates consists in initializing the environment randomly. As it will be emphasized later, it is interesting to have this option enabled so that the learning agents can be evaluated in different situations in each game.

ALE offers several ways to implement the agents:

- **Internal Interface**: The agents are compiled at the same time as ALE. Agents implemented by the user (in C++) may inherit a generic interface already coded by ALE developers.

- **Shared Library Interface**: The agents are compiled separately and communicate with ALE using the ALEInterface C++ object.

- **FIFO (First In First Out) Interface**: The agents communicate with ALE through the use of named pipes. It is independent of the programming language. An specific communication protocol must be followed.

- **RL-Glue Interface**: RL-Glue is a software module independent of ALE [26]. Its purpose is the standardization of an interface for reinforcement learning to ease code sharing. It works for several languages and platforms which support socket communications. As in FIFO, an specific communication protocol (RL-Glue 3.0 protocol) must be followed to execute the specified tasks.
Chapter 3

Methodology

3.1 Programming Language and Implementation Method

As specified in section 2.6, there are many languages with which we can develop agents that interact with ALE (C++, Python, Java, ...). Both reinforcement learning and MCTS methods require intensive computation and tend to be very time consuming. Thus, for this kind of tasks, it is recommended to use the C++ programming language rather than Python or Java [27].

Moreover, as explained before, the ALE interface is written in C++ and pipes are needed in case another language is used; therefore, since pipes might involve some delay in the interaction [28], C++ is the chosen language to implement the agents.

Furthermore, one of the implementation methods that ALE allows to use must be selected, so we have chosen the Shared Library Interface. This choice is made because it allows to easily implement the generalization part of this thesis: an ALEInterface object can be used for each game. By doing this, the agent can interact independently for each game in a very simple way (e.g. apply an action, get its last reward, get the current state of the game, ...).
3.2 Reinforcement Learning Agent

3.2.1 Algorithm Selection

As explained in chapter 1, the space of states in video games is usually vast. Atari games are not an exception.

In section 2.4, it was described that there were two possible ways to apply reinforcement learning: using a value function or function approximation. The first option requires to have each state-value pair in a table; therefore, since there are so many states, this option is not feasible and function approximation is used.

Besides, as we will see in section 3.4, function approximation allows generalization to be performed over more than one game. The reason is that each parameter $\theta(i)$ of the vector of parameters $\theta$ can generalize to states of different games if their features are represented using the same methodology.

There exists a kind of methods that use bootstrapping: they update an state-value or action-value from other state-values or action-values respectively. Sutton and Barto say in their book [15] that although bootstrapping methods do not have theoretical guarantees to perform better than non-bootstrapping methods, they usually do in practice. So bootstrapping methods are firstly considered.

The next important decision consists in selecting an algorithm that works with function approximation and bootstraps. The main criteria for selecting such method is that it neither diverges nor is unstable (in its basic form, without extensions). Therefore, there are two options which are immediately discarded: non-linear function approximation (e.g. using artificial neural networks) and any off-policy bootstrapping method.

Consequently, the remaining kind of methods are on-policy bootstrapping methods with linear function approximation. One of these methods is gradient-descent Sarsa with linear function approximation, which was previously described at section 2.4.3.
3.2.2 Feature Selection

The selection of gradient-descent Sarsa with linear function approximation implies that vectors of features $\phi$ must be used. ALE, as explained before, provides the screen being displayed as an array of pixels and the current state of the RAM as an array of bytes.

In case the screen was the chosen approach, an image processing algorithm should be used to get information of the screen (e.g. to know the objects present in the scene).

On the other hand, working with the RAM memory is much more straightforward since it is composed by bytes and, thus, by bits. Bits can be used directly as binary features, so almost no processing of the Atari RAM has to be done in order to create the feature vectors. An important characteristic of the RAM, without which it could not be used to produce the features (at least, it would not have much sense), is that it contains the game state and the score among other information (see section 2.1).

The first approach (which is used at algorithm 2) consists in a feature vector for each action $a$ given the current state $s$, $\phi_{s,a}$. Remember that the RAM of the system is 1024 bits long and 18 actions can be applied at any time in the game. Therefore, a method to build the feature vector for each action could consist of two parts: the 1024 bits of RAM and a binary identifier for each action. The identifier could be 18 binary digits such that if action $a \in [0, 17]$ was selected, then the $a^{th}$ digit of the sequence would be 1. The idea is graphically shown in figure 3.1.

![Figure 3.1: Feature vector $\phi_{s,a}$ with a common part (state $s$) and a variable part (action $a$)](image-url)
However, the previous approach has a problem which implies a bad behavior. The problem appears when the action-values are computed as the dot product between the parameter vector $\theta_t$ and the feature vector $\phi_{s,a}$:

$$Q_t(s, a) = \sum_{i=1}^{n} \theta_t(i) \phi_{s,a}(i)$$

As the state part has been set as common to all feature vectors, then only the action part is the one that makes the difference between the action-values $Q_t(s_t, a_t)$, so the same results could be achieved by removing all the state information.

The explained effect is not desirable because each time a greedy action selection is performed over the action-values, the selected action will be always that one which is generally greedy instead of the greedy one for the current state.

To solve that problem, there is a second approach which consists in having a parameter vector $\theta_a$ for each action $a$ and a single feature vector $\phi_s$. In fact, it is the inverse of the first method since in that one there was just one parameter vector $\theta$ and a feature vector $\phi_a$ for each action $a$. This approach was previously used in the Blocks World problem [29].

$$Q_t(s, a) = \sum_{i=1}^{n} \theta_{t,a}(i) \phi_{s}(i)$$

Since the feature vectors are no longer dependent on the actions, the action part of the first approach is removed. Therefore, only the part related to the RAM bits is maintained; in addition, apart from the 1024 positions (RAM bits), there is another single position added which always has value 1 (see figure 3.2).

![Feature vector $\phi_s$ with the state part got from the RAM and an extra position which is always 1](image)
This additional position is added because if all the RAM bits are 0, then the action-values will not be zero (unless all the positions of the parameter vector are 0). After some iterations in Sarsa, the corresponding parameter for this feature will be different than zero and the action-value will always be non-zero; therefore, if all features except the last are zero, the greedy action can also be determined.

### 3.2.3 Action-Values Normalization

During the Sarsa algorithm testing, the values of the function parameters increased its value extremely fast as soon as non-zero rewards started to be obtained. This happened to the point that these values could not be saved in the computer.

The cause that makes this happen is the number of non-zero features and the values of the function parameters. When non-zero rewards appear, the \( \delta \) variable of Sarsa and the function parameters start to become non-zero; consequently, the action-values are also non-zero. These action-values are computed from the dot product between the features and the function for an specific action; then, the corresponding action-value might have an enormous value after some iterations since it is used in the update of the function parameters (e.g. imagine that all features are 1 and all the function parameters are greater than 0).

The solution that has been used consists in normalizing the action-values since they are responsible for increasing the parameter values excessively. If they are normalized, the function parameters will not increase as fast as before. To normalize them, it is possible to divide the action-values by the number of non-zero features once they are computed [13]. This approach is the one used in our implementation of Sarsa.

### 3.2.4 Implementation

In the implementation of the reinforcement learning agent, there are two parts: the agent itself and the algorithm used by the agent. Both parts have been implemented using
the pattern of the ALE examples, which consists in dividing the interaction with the environment in three phases: start, step and end.

Algorithm 5 shows the pseudocode of the agent side. In each phase, the set of features $\phi_s$ is obtained from the current state $s$ using the method described in section 3.2.2.

Then, the process to follow is different depending on the phase being executed by the agent:

- In the start phase, an initial action is chosen and then applied to get the first reward of the episode.
- In the step phase, the process is the same as in the start but the action selection is done taking into account the last reward.
- In the end phase, the algorithm updates its parameters (if required) with the last reward and the current set of features.

Algorithm 5 Reinforcement Learning Agent with Function Approximation

```
1: Initialize $r \leftarrow 0$ \hfill ⤵ Last obtained reward
2: function EPISODESTART
3:  $\phi_s \leftarrow$ set of features of the current state $s$
4:  $a \leftarrow$ ALGORITHM.EPISODESTART($\phi_s$)
5:  $r \leftarrow$ ACT($a$)
6: end function

7: function EPISODESTEP
8:  $\phi_s \leftarrow$ set of features of the current state $s$
9:  $a \leftarrow$ ALGORITHM.EPISODESTEP($r, \phi_s$)
10: $r \leftarrow$ ACT($a$)
11: end function

12: function EPISODEEND
13:  $\phi_s \leftarrow$ set of features of the current state $s$
14:  ALGORITHM.EPISODEEND($r, \phi_s$)
15: end function
```

As explained before, the part that complements the agent is the algorithm. In the previous sections, it was determined that gradient-descent Sarsa with linear function approximation
using binary features would be the algorithm to use. The pseudocode of Sarsa is shown in algorithm 6 (especially note that the action-values are saved between phases).

The Sarsa pseudocode has been written taking into account all the decisions described in the previous sections: domain-independent optimistic initialization is applied along with the normalization of the action-values. Moreover, there is a function for each action and a single feature vector. Section 4.1.1.1 describes the selection of parameters for this algorithm.
Algorithm 6 Gradient-Descent Sarsa with Linear Function Approximation

1: Initialize $\theta_a \leftarrow 0, \forall a \in \mathcal{A}$
2: Initialize $Q_a \leftarrow 0, \forall a \in \mathcal{A}$
3: $r_{1st} \leftarrow$ first non-zero reward
4: $r_{1st-set} \leftarrow$ whether the $r_{1st}$ has been set (i.e. a non-zero reward has been seen)

5: function EpisodeStart($\phi_{s'}$)
6:   $r_{1st} \leftarrow 0$
7:   $r_{1st-set} \leftarrow$ false
8:   for all $b \in \mathcal{A}(s')$ do
9:       $Q_b \leftarrow \sum_i \theta_b(i)\phi_{s'}(i)$
10:      NormalizeActionValue($b, \phi_{s'}$)
11:   end for
12:   $a \leftarrow \pi(Q)$ \hspace{1cm} \triangleright \epsilon$-greedy policy
13:   $\phi_s \leftarrow \phi_{s'}$
14:   return $a$
15: end function

16: function EpisodeStep($r, \phi_{s'}$)
17:   SetOptimisticReward($r$)
18:   $r \leftarrow$ GetOptimisticReward($r$)
19:   with probability $1 - \epsilon$
20:     for all $b \in \mathcal{A}(s')$ do
21:         $Q_b \leftarrow \sum_i \theta_b(i)\phi_{s'}(i)$
22:         NormalizeActionValue($b, \phi_{s'}$)
23:     end for
24:     $a' \leftarrow \arg \max_{b \in \mathcal{A}(s')} Q_b$
25:   else
26:     $a' \leftarrow$ a random action $\in \mathcal{A}(s')$
27:     $Q_{a'} \leftarrow \sum_i \theta_{a'}(i)\phi_{s'}(i)$
28:     NormalizeActionValue($a', \phi_{s'}$)
29:   end with
30:   $\delta \leftarrow \delta + \gamma Q_{a'}$
31:   $\theta_a \leftarrow \theta_a + \alpha \delta \phi_s$
32:   $a \leftarrow a'; \phi_s \leftarrow \phi_{s'}$
33:   return $a$
34: end function

35: function EpisodeEnd($r, \phi_{s'}$)
36:   SetOptimisticReward($r$)
37:   $r \leftarrow$ GetOptimisticReward($r$)
38:   $r \leftarrow r - \gamma^{\text{numSteps(episode)}} - 1$
39:   $\delta \leftarrow r - Q_a$
40:   $\theta_a \leftarrow \theta_a + \alpha \delta \phi_s$
41: end function
Algorithm 6 Gradient-Descent Sarsa with Linear Function Approximation (continued)

42: function SetOptimisticReward(r)
43:     if ¬r_{1st-set} \land r > 0 then
44:         r_{1st-set} ← true
45:         r_{1st} ← r
46:     end if
47: end function

48: function GetOptimisticReward(r)
49:     if r_{1st-set} then
50:         r ← \frac{r}{r_{1st}} + γ - 1
51:     else
52:         r ← γ - 1
53:     end if
54:     return r
55: end function

56: function NormalizeActionValue(a, \phi_s)
57:     n ← Number of non-zero features in \phi_s
58:     Q_a ← \frac{Q_a}{n}
59: end function

3.3 Search Agent

3.3.1 Algorithm Selection

There exist diverse AI search methods that usually deal with games. Some of the possible methods that can be used are MCTS (e.g. UCT), minimax (with α – β pruning) or heuristic search (e.g. RTDP, LRTDP).

Minimax cannot be used in this context since it is used for adversarial games; thus, to derive the tree we do not only need to know our applicable actions, but also the opponent’s. For example, to apply minimax in Ms. Pac-Man, we would need to know how the ghosts behave, i.e. which actions they can perform at each time step. This information is not provided by the ALE framework, so this option is discarded.
On the other hand, heuristic search methods require the usage of an heuristic, while UCT does not. However, researchers often use heuristics in the UCT algorithm in some of its phases (e.g. selection or in the default policy). Since finding an heuristic can be difficult (and even more if it must generalize to several games), UCT is a better option in our case.

### 3.3.2 Implementation

An indispensable feature has to be provided by the ALE framework to implement UCT (in fact, any search algorithm): ability to save and load states. As explained in section 2.6, the actions 43 (save-state) and 44 (load-state) give the required functionalities.

This feature is necessary since when performing simulations (default policy phase), the actual current state of the game is lost (i.e. the state over which the decided action had to be applied). Thus, before performing a simulation, the current state of the game is saved; then, the simulation is run, and thereafter the previously saved state is loaded again.

Algorithm 4 indicates that some computational budget has to be specified in the function UCTSearch($s_0$). The decision is to set a limit number of iterations, which is the usual approach.

After some first tests, it was clear that some other restrictions had to be implemented in the UCT algorithm because of time execution issues. The epicenter that causes big delays in execution is the simulation phase (i.e. default policy): many times each action application takes more than a second. Consequently, a possible solution consists in setting a maximum number of frames to simulate; thus, if either a terminal state is reached or the maximum number of frames is exceeded, the simulation stops.

Another addition to the UCT algorithm consists in a discount factor $\gamma$ as in reinforcement learning. During the preliminary tests with Ms. Pac-Man, it was observed that when the game was nearly finished (e.g. one remaining sweet to eat), the character started to oscillate around the sweet without eating it. A possible cause of this behavior may be that several actions can lead to the same amount of reward and, thus, a clear path to the
sweet cannot be set. Besides, it is also possible that the number of simulated frames is not enough to reach any of the sweets, so all actions would lead to no reward.

Figure 3.3 shows an example of the oscillations: any action can be used to get the sweet in the right part of the screen (i.e. identical reward). For example, the character could go directly left or go right-up-down-up-left (among many other combinations), and it would obtain the same reward. Therefore, if the 18 possible actions have equal benefit, then it can be difficult that Pacman reaches the sweet.

![Figure 3.3: Example of oscillation towards reward in Ms. Pac-Man](image)

The previously mentioned discount factor can be applied in the backpropagation phase. The propagated reward would be multiplied by 1 in the first traversed node, by $\gamma$ in the second, by $\gamma^2$ in the third, and so on until the root is reached. By doing this, shorter paths to a reward are favored respect to the longer ones.

An improvement to save memory resources is done after selecting the best child. Since the only part over which the following iterations will be done is the subtree whose root is the best child, all the other parts of the tree can be pruned (see figure 3.4). Therefore, a new tree will not be created each time a search is to be performed.
Another way to save resources consists in updating the average value each time a node is visited during backpropagation (sum of the values could even cause overflow). The update rule would change as follows:

\[
N(v) \leftarrow N(v) + 1 \\
Q(v) \leftarrow Q(v) + \frac{\Delta - Q(v)}{N(v)}
\]

To implement the search agent, the same pattern used in the reinforcement learning agent is adopted (see algorithm 7). That is, the agent has three routines: start, step and end. Since the tree is pruned at each step, the root needs only to be created once; for this reason, a new function called \texttt{InitializeTree(s0)} is used to create the tree with the starting state of the game. This function is only called in the \texttt{EpisodeStart} routine since in the following iterations, the new root will be automatically set by UCT when pruning.

Besides, we have seen that applying an action \(a\) in a state \(s\) always results in the same state \(s'\); if it did not happen this way, we should initialize the tree at each step and not only at the beginning. Nevertheless, in the code it was compared at each step if the state of the tree’s root was the same as the current one controlled by the agent; thus, if they are different, a new call to \texttt{InitializeTree(s0)} can be performed in order to solve this incongruence.

The agent performs a search at each step (including the start phase) to get the presumable best action. Then it applies this action to the environment to get the reward (however,
in this case, the reward is not required by the algorithm). There is not end phase in this
agent since any model needs to be updated.

Algorithm 8 shows the UCT algorithm with the modifications described in the previous
paragraphs; especially note that there are calls to \texttt{SaveState}(s(v)) and \texttt{LoadState}
before and after a simulation respectively. The selection of the UCT parameters is described
in section 4.1.1.2.

\begin{algorithm}
\caption{Search Agent using the UCT algorithm}
\begin{algorithmic}
\State \textbf{Initialize }$r \leftarrow 0$ \Comment{Last obtained reward}
\Function{EpisodeStart} \endComment
\State $s_0 \leftarrow \text{InitializeTree}(s_0)$ \Comment{$s_0$ is the initial state of the tree}
\State $a \leftarrow \text{UctSearch}$
\State $r \leftarrow \text{Act}(a)$
\EndFunction
\Function{EpisodeStep} \endComment
\State $a \leftarrow \text{UctSearch}$
\State $r \leftarrow \text{Act}(a)$
\EndFunction
\end{algorithmic}
\end{algorithm}

### Implementing RAVE

In this section it is explained how a tree policy variant, RAVE, for MCTS methods is
implemented (see section 2.5.4).

The main feature of RAVE (and AMAF methods in general) consists in updating not only
the selected nodes during the tree policy, but also some of their neighbors. These neighbors
are those that have an state equal to another found during the simulations (default policy).

The first approach followed was exactly the same as the one explained in section 2.5.4.
However, there was a problem: a state found during the simulations never matched a state
in the tree.

The ALE framework provides a method for comparing states, however, it compares the
current frame in the simulation and a serialized string that represents information of the
Algorithm 8 Modified UCT (based on algorithm 4)

1: function $\text{InitializeTree}(s_0)$
2: create root node $v_0$ with state $s_0$
3: end function

4: function $\text{UctSearch}$
5: $i \leftarrow 0$
6: while $i <$ maximum number of iterations do
7: $v_l \leftarrow \text{TreePolicy}(v_0)$
8: $\Delta \leftarrow \text{DefaultPolicy}(s(v_l))$
9: $\text{Backup}(v_l, \Delta)$
10: $i \leftarrow i + 1$
11: end while
12: $v_b \leftarrow \text{BestChild}(v_0, 0)$
13: $\text{PruneTree}(v_0, v_b)$
14: return $a(v_b)$
15: end function

16: function $\text{TreePolicy}(v)$
17: while $v$ is nonterminal do
18: if $v$ not fully expanded then
19: return $\text{Expand}(v)$
20: else
21: $v \leftarrow \text{BestChild}(v, C_p)$
22: end if
23: end while
24: return $v$
25: end function

26: function $\text{BestChild}(v, c)$
27: return $\text{arg max}_{v' \in \text{children of } v} Q(v') + c \sqrt{\frac{2 \ln N(v)}{N(v')}}$
28: end function

29: function $\text{Expand}(v)$
30: $\text{SaveState}(s(v))$
31: choose $a \in \text{untried actions from } \mathcal{A}(s(v))$
32: add a new child $v'$ to $v$ with $s(v') = f(s(v), a)$ and $a(v') = a$
33: $\text{LoadState}$
34: return $v$
35: end function
Algorithm 8 Modified UCT (continued)

35: function DefaultPolicy(s)
36:      SaveState(s(v))
37:      while s is non-terminal do
38:         choose a ∈ A(s) uniformly at random
39:         s ← f(s, a)
40:      end while
41:      LoadState
42:      return reward for state s
43:   end function

44: function Backup(v, ∆)
45:      n ← 0
46:      while v is not null do
47:         N(v) ← N(v) + 1
48:         Q(v) ← Q(v) + γ^nΔ(v,p)−Q(v)
49:         v ← parent of v
50:         n ← n + 1
51:   end while
52: end function

53: function PruneTree(v₀, vᵇ)
54:      erase all v₀ and all its subtrees except for vᵇ
55:      v₀ ← vᵇ
56: end function

Game. Since the frame number of the simulated states would always be higher than the one of the state in the tree, states were always different. Therefore, another method to compare only the serialized strings was created, but any simulated state was equal to a tree state, so the performance was extremely poor.

To overcome the previous problem, a simplified version of RAVE is implemented: all the neighbors of a node in the backpropagation path have their AMAF counters updated. Thus, it is equivalent to assume that all neighbor states will eventually be found during the simulations.

Algorithm 9 shows the modified functions to implement RAVE. The UCT variables for visits and average value are denoted by N(v) and Q(v) respectively, while the AMAF-RAVE counters are Nₐ(v) and Qₐ(v). The RAVE parameter to determine the α is called
Methodology

Algorithm 9 UCT routines implementing RAVE (based on algorithm 8)

1: function BestChild(v, c)
2: return \( \arg \max_{v' \in \text{children of } v} \alpha \cdot c_{\text{amaf}} + (1 - \alpha) \cdot c_{\text{uct}} \)
3: \( c_{\text{uct}} \leftarrow Q(v') + c \sqrt{\frac{2 \ln N(v')}{N(v')}} \)
4: \( c_{\text{amaf}} \leftarrow Q_R(v') + c \sqrt{\frac{2 \ln N_R(v')}{N_R(v')}} \)
5: \( \alpha \leftarrow \max \left( \frac{rave_{\text{param}} - N_R(v')}{rave_{\text{param}}}, 0 \right) \)
6: end function

7: function Backup(v, \Delta)
8: \( n \leftarrow 0 \)
9: while v is not null do
10: \( N(v) \leftarrow N(v) + 1 \)
11: \( Q(v) \leftarrow Q(v) + \frac{\gamma^n \Delta(v, p) - Q(v)}{N(v)} \)
12: \( N_R(v) \leftarrow N_R(v) + 1 \)
13: \( Q_R(v) \leftarrow Q_R(v) + \frac{\gamma^n \Delta(v, p) - Q_R(v)}{N_R(v)} \)
14: for all siblings \( v_s \) of \( v \) do
15: \( N_R(v_s) \leftarrow N_R(v_s) + 1 \)
16: \( Q_R(v_s) \leftarrow Q_R(v_s) + \frac{\gamma^n \Delta(v_s, p) - Q_R(v_s)}{N_R(v_s)} \)
17: end for
18: \( v \leftarrow \text{parent of } v \)
19: \( n \leftarrow n + 1 \)
20: end while
21: end function

3.4 Multiple Game Learning Agent

One of the main topics in this thesis consists in developing an agent that learns in multiple games simultaneously. Humans tend to build good generalizations of games, so it is interesting to test if learning algorithms can do it as well. For example, a human could build a generalization from the games Ms. Pac-Man and Alien by focusing in the game mechanics and visualizing them (see figure 3.5); some of the common aspects between these games are:

- The map is a maze.
Methodology

Figure 3.5: Screenshots of Ms. Pac-Man and Alien

- There are some "sweets" that increase our score when the controlled character reaches their position.
- There are monsters that can kill us if the controlled character touches them.
- There are special "sweets" which give the ability to kill the monsters to score more points.
- The game or phase ends when all "sweets" are collected.
- There are doors which allow the character to go directly from one extreme of the map to the other.

The learning process is done by using the previously explained gradient-descent Sarsa with linear function approximation. Actually, function approximation allows us to do this kind of tests easily since all games are represented by the same features (RAM bits); the difference is that in this case a function does not generalize to one game, but to several games simultaneously. On the other hand, search-based methods do not allow us to perform this kind of experiments: different games might be described by different states, so building a tree of states of diverse games does not have much sense. Function approximation offers a common way to store information, so it is a better method to perform the tests.

The main idea consists in performing a learning step in each game sequentially. That is, given two games $g_1$ and $g_2$, a learning step can be done on $g_1$ and, after that, another step would be done in $g_2$. 
Two kinds of learning steps are considered:

- **Action steps**: The agent applies one action to each game one after another.

- **Episode steps**: The agent executes one episode of each game one after another.

Action steps have a big drawback: the length of the episodes for each game might be different. Therefore, a criteria must be defined to set when the games must be restarted (i.e. each game starts a new episode). The are two distinct possibilities:

1. Restart when some game ends its episode.

2. Restart when all games finish their episodes.

In the first case (1), there may be always a game which finishes its episode while the other one does not. Therefore, the model can be mostly based on one game because the other one does not always finish its episode.

Something similar to (1) happens in (2). Although in this case all games finish their episodes, it is possible that one of them ($g_1$) finishes much earlier than another one ($g_2$). Therefore, $g_1$ might have little influence on the model, so the function is mainly built from $g_2$ data.

Note that in both (1) and (2) we have affirmed that the game that executes more episodes might be the one with more influence in the model. It is hypothetical since it should also be the game which usually scores more points during the episode; that is, if a game $g_1$ executes for more episodes and score less points than $g_2$, then it may not influence the model as much as $g_2$.

Furthermore, if action steps were used, it is important to remember that Sarsa requires to know which was the last action used and also the last feature vector (in case the action-values are not saved between steps). Therefore, if two games are used during the learning process, then we should define which actions and features are saved between steps:
1. Save the last action and last feature vector for each game.

2. Save the last action and last feature vector independently of the last used game.

We consider that it would not be convenient to mix actions and feature vectors between steps because it could be interpreted as playing a game made from other games (which is not the case). So if action steps were used, it is believed that the approach having separate actions and features for each game would be a better approach.

Episode steps also suffer from the episodes length problem: a game whose episodes are longer and that obtains positive reward more frequently might have more influence on the function being created.

The order in which game selection is done also matters. It is not the same completing an episode for $g_1$ and then for $g_2$ than in the reverse way (the same applies to action steps). Three selection methods are implemented:

- **Ascending Order**: The games specified in the configuration file are selected in ascending order successively. If there are three games: $g_1, g_2, g_3, g_1, \ldots$

- **Descending Order**: The games specified in the configuration file are selected in descending order successively. If there are three games: $g_3, g_2, g_1, g_3, \ldots$

- **Random**: The games specified in the configuration file are selected randomly. For example: $g_1, g_1, g_3, g_1, g_2, \ldots$

The selection of a game is done in different moments depending on whether action or episode steps are used. In the case of action steps, the game selection is done before choosing an action to apply; on the other hand, in episode steps, the selection is done before starting an episode.

Another problem of learning from multiple games simultaneously is that the reward scale of each game can be different. Consequently, some way to normalize them to a common scale is needed. A simple way to do it consists in transforming all rewards greater than 0
Methodology

to 1, and all rewards lower than 0 to -1. It could be also possible to divide any reward by
the maximum achievable reward during the game, but this maximum reward is not known
in advance.

Algorithm 10 represents how the agent works with multiple games. There is only a small
addition respect to algorithm 5: calls to SELECTGAME are done in diverse parts of the
algorithm depending on the kind of learning step used (read the comments of the right
part in the algorithm). Besides, the reward normalization is also added with the function
NORMALIZE_REWARD. Note that all the routines related to the ALGORITHM are the ones
in algorithm 6.

In section 4.2.1 the selection of the previously explained parameters is done to perform
the corresponding experiments.
Algorithm 10 Multiple Game Learning Agent (modification of algorithm 5)

1: Initialize $r \leftarrow 0$  \hfill \triangleright Last obtained reward

2: function EpisodeStart
3:     SelectGame  \hfill \triangleright Episode Steps or Action Steps
4:     $\phi_s \leftarrow$ set of features of the current state $s$
5:     $a \leftarrow$ ALGORITHM.EpisodeStart($\phi_s$)
6:     $r \leftarrow$ Act($a$)
7:     NormalizeReward
8: end function

9: function EpisodeStep
10:     SelectGame  \hfill \triangleright Action Steps
11:     $\phi_s \leftarrow$ set of features of the current state $s$
12:     $a \leftarrow$ ALGORITHM.EpisodeStep($r, \phi_s$)
13:     $r \leftarrow$ Act($a$)
14:     NormalizeReward
15: end function

16: function EpisodeEnd
17:     $\phi_s \leftarrow$ set of features of the current state $s$
18:     ALGORITHM.EpisodeEnd($r, \phi_s$)
19: end function

20: function NormalizeReward
21:     if $r > 0$ then
22:         $r \leftarrow 1$
23:     end if
24:     if $r < 0$ then
25:         $r \leftarrow -1$
26:     end if
27: end function
Chapter 4

Evaluation

4.1 Learning in a Single Game

4.1.1 Parameter Selection

An essential task when using learning algorithms consists in setting the appropriate parameters to get the best possible results. During the implementation, we specified two kinds of configuration attributes:

- **General attributes**: These attributes are common to all algorithms. For example: the frame skip or the maximum number of frames per episode.

- **Algorithm attributes**: These attributes depend exclusively on the algorithm used. For example: number of iterations in UCT or the step-size ($\alpha$) parameter in Sarsa.

The algorithm attributes are explained in the upcoming sections, while the general ones are detailed below.

The first parameter specified in all configurations of the agents is the **number of episodes**. Although it is general to any agent, the values of it change depending on the agent:
- **Reinforcement Learning Agent:** During the training process, 20000 or 30000 episodes are used. On the other hand, the evaluation of a specific function is done with 1000 episodes since they do not involve learning.

- **Search Agent:** This agent does not require a training phase, only 5 or 10 episodes are used for each game since the execution time required can be much higher than with Sarsa.

Another parameter related to the duration of the simulation are the **maximum number of frames per episode** which is always set to 15000 frames; that is, approximately playing for 4 minutes since games are run at 60 frames per second. Thus, as soon as the game reaches the 15000 frames of execution, the current episode is stopped and a new episode starts.

A major drawback of setting a limit of frames is that the agent might not be able to get all the score it is capable of; on the other hand, it is necessary to set a limit because of time issues, specially in UCT which requires a lot of time to complete an episode (depending on the game). The main cause for which UCT is sometimes slow, is that the emulator usually requires about a second to apply an action; therefore, the accumulated amount of seconds results in a long execution time.

The **random initialization** of the environment explained in 2.6 is also used. We think that it is especially needed in reinforcement learning methods because without this parameter, the agent would always learn a good sequence of actions for a single situation but not for many (which is the main purpose of an AI agent). It is not so important in UCT since a model or function is not saved used between episodes; instead, the search tree is built again without taking into account any past information. However, it is interesting to keep it enabled for UCT to know how it works given different situations.

The last ALE parameter used is the **frame skip**: the number of frames during which an action is applied. In this case, an amount of 10 frames per action is fixed. The reason why 10 frames per action are used is because in the game Ms. Pac-Man, approximately 10 frames are skipped between each sweet position of the character controlled by the player.
Frame skip has been shown to be a determining factor for obtaining good results in games [30]. If it is assigned a very high value, many important steps in which huge amounts reward can be obtained are missed. For example, having a high frame skip in Ms. Pac-Man might lead to bad performance since the character can be blocked in a wall (in case the number of frames required to reach that wall is less than the frame skip), so opportunities to get some points by following other itineraries are lost.

On the other hand, if the frame skip is low, then the agent performs with better resolution, which specially fits games like Breakout or Pong, where precision on hitting the ball is extremely important to get high rewards.

However, not only low frame skip is beneficial. Braylan et. al [30] show that games such as Seaquest or Space Invaders perform well with high frame skips (180 and 60 frames respectively).

Finally, the entire set of actions (18) is used in all implemented methods. That is, at each time step one action of the full set of actions is selected instead of the minimal set of actions (i.e. those actions which are useful in the game). Since we want the agents to discover which actions are useful or not in each case, it is decided that all of them will have to deal with all possible actions.

The appropriate value of a parameter varies depending on the game (as we have seen in the frame skip). To get those parameter values that are beneficial for the performance of each algorithm, some tests are done with diverse combinations of them using a common training set of 4 games: Alien, Atlantis, Ms. Pac-Man and Pooyan.

4.1.1.1 Gradient-Descent Sarsa Parameters

Gradient-descent Sarsa with linear function approximation is a learning algorithm which has several parameters: $\alpha$ (step-size), $\epsilon$ (exploration), and $\gamma$ (bootstrapping). Moreover, optimistic initialization can be also used to encourage exploration.
The **discount factor** parameter, $\gamma$, has been assigned a value of 0.999. Values that are close to 1 are often used for this parameter [13, 16]. Therefore, the action-value of the next state-action pair, $Q(s_{t+1}, a_{t+1})$, gets great importance when updating the corresponding parameter vector.

There are three parameters which have not a clear value: $\alpha$, $\epsilon$ and the optimistic initialization. The set of values to be tested for each parameter are: $\alpha \in \{0.01, 0.05, 0.1, 0.2\}$, $\epsilon \in \{0.01, 0.05, 0.1, 0.2\}$, and optimistic initialization can be either true or false.

A total of 128 experiments are run for 20000 episodes with the general parameters specified in section 4.1.1. Once they are all executed, the 10 combinations with highest score in the last 1000 episodes are ranked for each game (see table 4.1).

Once the table is built, we require a method to discriminate the best values from the worst ones. To do that, the average position is computed for each value of each parameter; the lower the average is, the better that value becomes for the problem. Table 4.2 shows the position of each game for each combination; this helps to see which combinations of parameters are usually good for the training games.

From the table 4.2 is easy to compute the previously mentioned position averages:

$$\text{avg}_{\alpha=0.01} = \frac{4}{1} = 4$$

$$\text{avg}_{\alpha=0.05} = \frac{61}{10} = 6.1$$

$$\text{avg}_{\alpha=0.1} = \frac{77}{14} = 5.5$$

$$\text{avg}_{\alpha=0.2} = \frac{78}{15} = 5.2$$
<table>
<thead>
<tr>
<th>Game</th>
<th>Rank</th>
<th>Score (last 1000 episodes)</th>
<th>Overall Score</th>
<th>$\alpha$</th>
<th>$\epsilon$</th>
<th>Optimistic Initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alien</td>
<td>1</td>
<td>2111.16</td>
<td>1249.52</td>
<td>0.1</td>
<td>0.01</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1630.36</td>
<td>1044.04</td>
<td>0.1</td>
<td>0.01</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1226.63</td>
<td>1020.56</td>
<td>0.2</td>
<td>0.01</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1200.84</td>
<td>850.946</td>
<td>0.2</td>
<td>0.01</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1083.52</td>
<td>896.14</td>
<td>0.2</td>
<td>0.05</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>987.02</td>
<td>818.959</td>
<td>0.2</td>
<td>0.05</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>986.25</td>
<td>843.79</td>
<td>0.1</td>
<td>0.05</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>919.90</td>
<td>806.74</td>
<td>0.1</td>
<td>0.05</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>789.9</td>
<td>650.534</td>
<td>0.05</td>
<td>0.05</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>761.27</td>
<td>731.011</td>
<td>0.05</td>
<td>0.1</td>
<td>false</td>
</tr>
<tr>
<td>Atlantis</td>
<td>1</td>
<td>27587.1</td>
<td>20756.2</td>
<td>0.05</td>
<td>0.01</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>25983.7</td>
<td>18442.9</td>
<td>0.05</td>
<td>0.01</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>24665.7</td>
<td>26287.4</td>
<td>0.2</td>
<td>0.01</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>24492.7</td>
<td>22100.7</td>
<td>0.2</td>
<td>0.01</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>22868.2</td>
<td>20378.4</td>
<td>0.05</td>
<td>0.05</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>20127.3</td>
<td>18519.1</td>
<td>0.1</td>
<td>0.01</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>19847.7</td>
<td>16790.5</td>
<td>0.2</td>
<td>0.05</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>19222.4</td>
<td>20851.6</td>
<td>0.1</td>
<td>0.01</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>16441.7</td>
<td>15921.2</td>
<td>0.2</td>
<td>0.03</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>16466.6</td>
<td>14792.5</td>
<td>0.05</td>
<td>0.2</td>
<td>false</td>
</tr>
<tr>
<td>Ms. Pac-Man</td>
<td>1</td>
<td>1755.13</td>
<td>1415.87</td>
<td>0.2</td>
<td>0.01</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1541.92</td>
<td>1186.16</td>
<td>0.2</td>
<td>0.01</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1523.91</td>
<td>1086.16</td>
<td>0.2</td>
<td>0.01</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1408.59</td>
<td>1098.63</td>
<td>0.2</td>
<td>0.05</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1233.37</td>
<td>980.269</td>
<td>0.1</td>
<td>0.05</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>1227.34</td>
<td>986.554</td>
<td>0.2</td>
<td>0.05</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>1158.88</td>
<td>890.142</td>
<td>0.1</td>
<td>0.01</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1130.38</td>
<td>890.107</td>
<td>0.2</td>
<td>0.1</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>1117.78</td>
<td>875.13</td>
<td>0.05</td>
<td>0.05</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>1111.1</td>
<td>966.04</td>
<td>0.2</td>
<td>0.1</td>
<td>true</td>
</tr>
<tr>
<td>Pooyan</td>
<td>1</td>
<td>1339.895</td>
<td>1186.19</td>
<td>0.05</td>
<td>0.01</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1327.14</td>
<td>1058.56</td>
<td>0.1</td>
<td>0.01</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1301.535</td>
<td>1116.7</td>
<td>0.1</td>
<td>0.01</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1234.685</td>
<td>686.103</td>
<td>0.01</td>
<td>0.05</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1210.12</td>
<td>1127.77</td>
<td>0.2</td>
<td>0.01</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>1205.65</td>
<td>1138.99</td>
<td>0.05</td>
<td>0.05</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>1183.015</td>
<td>1003.46</td>
<td>0.1</td>
<td>0.05</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1182.065</td>
<td>996.729</td>
<td>0.05</td>
<td>0.05</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>1143.505</td>
<td>1128.38</td>
<td>0.1</td>
<td>0.05</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>1141.705</td>
<td>984.509</td>
<td>0.1</td>
<td>0.1</td>
<td>false</td>
</tr>
</tbody>
</table>

Table 4.1: Best combinations of the Sarsa parameters ($\alpha$, $\epsilon$ and optimistic initialization) for the training games

\[
\text{avg}_{\epsilon=0.01} = \frac{56}{17} \approx 3.2941
\]

\[
\text{avg}_{\epsilon=0.05} = \frac{116}{18} = 6.4
\]
<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$\epsilon$</th>
<th>Optimistic Initialization</th>
<th>Positions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Alien</td>
</tr>
<tr>
<td>0.01</td>
<td>false</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>true</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.05</td>
<td>false</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>true</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.1</td>
<td>false</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>true</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.2</td>
<td>false</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>true</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.01</td>
<td>false</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>true</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.05</td>
<td>false</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>true</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.1</td>
<td>false</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>true</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.2</td>
<td>false</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>true</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.01</td>
<td>false</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>true</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.05</td>
<td>false</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>true</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.1</td>
<td>false</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>true</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.2</td>
<td>false</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>true</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Game ranking positions for all the combinations of $\alpha$, $\epsilon$ and optimistic initialization

$$\text{avg}_{\epsilon=0.1} = \frac{38}{4} = 9.5$$

$$\text{avg}_{\epsilon=0.2} = \frac{10}{1} = 10$$
\[
\text{avg}_{\text{opt=true}} = \frac{105}{20} = 5.25
\]

\[
\text{avg}_{\text{opt=false}} = \frac{115}{20} = 5.75
\]

From the averages, the following conclusions are obtained:

- **Step-Size Parameter (\(\alpha\))**: Although 0.01 is the option with the best average, it has been used only once; therefore, its usage with other games might not be reliable. In consequence, the main options are 0.1 and 0.2, whose averages are very similar and they have been visited almost the same number of times.

- **Exploration Parameter (\(\epsilon\))**: It is clear that 0.01 is the best option since its average position is the best one with a relatively big difference over the second best option (0.05). Besides, 0.01 appears approximately as many times as 0.05 in the ranking.

- **Optimistic Initialization**: The difference between using optimistic initialization or not is not very high. Moreover, both options are chosen the same number of times and it is not clear whether it is better to enable it or not.

Table 4.1 shows that optimistic initialization is usually an enhancement for \(\alpha\) and \(\epsilon\) since, in a training game, the same pair \(< \alpha, \epsilon >\) tends to appear twice: one with optimistic initialization and one without it. For example, the first 8 positions for Alien are consecutive \(< \alpha, \epsilon >\) pairs with and without optimistic initialization. Concretely there are 4 pairs for Alien, Atlantis and Ms. Pac-Man, and 3 pairs for Pooyan.

From the previous analysis, only the best value for \(\epsilon\) is known. On the other hand, the value of \(\alpha\) and whether enabling or not optimistic initialization are not clear. For this reason, another table is created by fixing \(\epsilon = 0.01\) and varying \(\alpha\) and the optimistic initialization. However, in this case what is compared is how good combinations are respect to the best combination for each game (see table 4.3).
Evaluation

<table>
<thead>
<tr>
<th>Game</th>
<th>$\alpha$</th>
<th>Optimistic Initialization</th>
<th>Score (last 1000 episodes)</th>
<th>Best Score (Last 1000 episodes)</th>
<th>Times better</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alien</td>
<td>0.1</td>
<td>false</td>
<td>1640.36</td>
<td>2111.16</td>
<td>0.7723</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>false</td>
<td>1226.3</td>
<td>1248.84</td>
<td>0.6588</td>
</tr>
<tr>
<td></td>
<td></td>
<td>true</td>
<td>1922.4</td>
<td>27578.1</td>
<td>0.7298</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>true</td>
<td>2492.7</td>
<td>2665.7</td>
<td>0.8881</td>
</tr>
<tr>
<td>Atlantis</td>
<td>0.1</td>
<td>false</td>
<td>1301.555</td>
<td>1339.895</td>
<td>0.9904</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>false</td>
<td>1240.12</td>
<td>1755.13</td>
<td>0.8155</td>
</tr>
<tr>
<td></td>
<td></td>
<td>true</td>
<td>968.74</td>
<td>1158.88</td>
<td>0.9031</td>
</tr>
<tr>
<td>Pooyan</td>
<td>0.1</td>
<td>false</td>
<td>1327.14</td>
<td>1491.90</td>
<td>1.0000</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>false</td>
<td>1521.91</td>
<td>1755.13</td>
<td>1.0000</td>
</tr>
<tr>
<td>Ms. Pac-Man</td>
<td>0.1</td>
<td>false</td>
<td>1301.555</td>
<td>1339.895</td>
<td>0.9904</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>false</td>
<td>1240.12</td>
<td>1755.13</td>
<td>0.8155</td>
</tr>
<tr>
<td></td>
<td></td>
<td>true</td>
<td>968.74</td>
<td>1158.88</td>
<td>0.9031</td>
</tr>
</tbody>
</table>

Table 4.3: Relative scores to the highest scores for the training games with $\epsilon = 0.01$

Once the relationship number between the score of each combination with $\epsilon = 0.01$ and the best score for each game are obtained, the average of each combination is computed. The higher the relative score is, the better combination it is.

$$\text{avg}_{\alpha=0.1, \text{opt}=\text{false}} = \frac{0.7723 + 0.697 + 0.9904 + 0.5508}{4} \approx 0.7526$$

$$\text{avg}_{\alpha=0.1, \text{opt}=\text{true}} = \frac{1 + 0.7298 + 0.9714 + 0.6603}{4} \approx 0.8404$$

$$\text{avg}_{\alpha=0.2, \text{opt}=\text{false}} = \frac{0.5809 + 0.8881 + 0.9031 + 0.8683}{4} \approx 0.8101$$

$$\text{avg}_{\alpha=0.2, \text{opt}=\text{true}} = \frac{0.5688 + 0.8944 + 0.8155 + 1}{4} \approx 0.8197$$

As shown in the previous equations, the best combination is $\alpha = 0.1$, $\epsilon = 0.01$ and optimistic initialization enabled. These parameters will be used in upcoming sections with the entire set of games.
**Sensibility to the Learning Parameters**

During the selection of the reinforcement learning parameters, it could be seen that each game tends to work well with an specific combination while with another one, its results are much lower. In this section, the parameters analyzed before ($\alpha$, $\epsilon$ and optimistic initialization) will be compared for each game through the learning curves. Each point of a curve corresponds to the average score of the last 1000 episodes.

Firstly, in figure 4.1, a graphic of the tested values of $\alpha$ is plotted for each game. The parameter $\epsilon$ is fixed to 0.01 and the optimistic initialization is enabled. As it can be observed, each game reacts differently to each value of $\alpha$, although they do not often perform well with $\alpha = 0.01$ (specially in Atlantis and Pooyan). On the other hand, when $\alpha \in \{0.05, 0.1, 0.2\}$ is used, the results tend to be much better. There are cases in which it is clear that an specific $\alpha$ is better than another for some games; for example, $\alpha = 0.2$ in Ms. Pac-Man, $\alpha = 0.1$ in Alien or $\alpha = 0.05$ in Atlantis. Besides, in some games the difference between two or more values of $\alpha$ does not differ much in the final average result (for instance, Pooyan performs similarly with $\alpha = 0.05$ and $\alpha = 0.1$).

With reference to the forms of the graphics, the learning process usually has moments during which significant unlearning takes place (i.e. the slope of the curve becomes negative). It mostly happens with $\alpha = 0.2$ (e.g. Alien, Atlantis, Pooyan), and rarely recovers from it. A reason that might cause this problem is that $\alpha = 0.2$ makes the action-value to oscillate around its true value since the updates are too big. A possible solution to the last problem could be to decrease $\alpha$ while learning; it might help to avoid the unlearning phases because large updates would not be done, so if good behavior is achieved, it can be maintained since the variations become smaller with time.

In figure 4.2, there are some graphics with variable $\epsilon$, whereas $\alpha$ is 0.1 and optimistic initialization is enabled. The value of $\epsilon$ that gives the best results is 0.01 by a wide margin in Alien and Atlantis. The second best value is $\epsilon = 0.05$, which achieves the best result in Ms. Pac-Man. On the other hand, the highest tested values (0.1 and 0.2), do not often perform as well as the best values of $\epsilon$ (except for Pooyan). Generalizing to these 4 games,
it is clear that the best values for exploration are usually small; therefore, it means that too much exploration during learning is not beneficial (i.e. it is better to act greedily and explore few times).

Optimistic initialization graphics are shown in figure 4.3. The exploration parameter $\epsilon$ has been set to 0.01, while the $\alpha$ and the optimistic initialization values have been combined to show two cases of optimistic initialization in each game.

In Ms. Pac-Man, optimistic initialization works well for both $\alpha$ values used (0.1 and 0.2) since it improves the average score in both cases. This improvement also appears in Alien when $\alpha = 0.1$.

However, using optimistic initialization does not always result in better results. In Atlantis, for example, the score difference between being optimistic or not is very similar; the same happens with Alien using $\alpha = 0.2$. Besides, in Pooyan, agents that are not optimistic perform better than those who are; a possible cause for this behavior could be that exploratory actions were usually not beneficial for achieving a higher score (remember that optimistic initialization is used to encourage exploration during the learning process).
Figure 4.2: Learning curves of training games with variable exploration (ε) values

Figure 4.3: Learning curves of training games with and without optimistic initialization
4.1.1.2 Monte Carlo Tree Search Parameters

To achieve the best possible results in a determined amount of time, some parameters of the UCT algorithm (the chosen search algorithm in section 3.3.1) must be configured.

The discount factor is set to 0.999 as in the reinforcement learning algorithm. Since our purpose was to favor the shortest paths to a reward, having a discount factor of 0.999 is enough.

As commented in section 3.3.2, there are two parameters which can be set to adjust the time of execution: the number of iterations per node and the number of simulated frames. The first parameter determines the depth of the tree being built, and the second one for how many frames random actions are selected. The higher value they have, the higher the amount of time to run the algorithm is. However, if the values of both parameters are high, then the results will be probably better because the game is further explored.

Then, to choose a value for the previous parameters, some combinations of them are executed in Ms. Pac-Man. Firstly, the time required to complete one search for each combination is evaluated; if the selected combination runs in 20 seconds or less, then the average score for 5 episodes is compared to get the winning combination. Table 4.4 shows some tested combinations setting the exploration constant $c = 0.15$; the combination with the highest score is chosen for the next experiments (175, 200).

<table>
<thead>
<tr>
<th>Iterations per node</th>
<th>Simulated frames</th>
<th>Time/Search (seconds)</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>100</td>
<td>18</td>
<td>3322</td>
</tr>
<tr>
<td>250</td>
<td>200</td>
<td>28</td>
<td>-</td>
</tr>
<tr>
<td>250</td>
<td>150</td>
<td>18</td>
<td>4378</td>
</tr>
<tr>
<td>200</td>
<td>200</td>
<td>23</td>
<td>-</td>
</tr>
<tr>
<td>150</td>
<td>200</td>
<td>17</td>
<td>4252</td>
</tr>
<tr>
<td>175</td>
<td>200</td>
<td>20</td>
<td>4668</td>
</tr>
</tbody>
</table>

Table 4.4: Evaluation of time and score for some combinations of the number iterations per node and the number of simulated frames in Ms. Pac-Man
The next step consists in determining the best exploration constant since all the other parameters have already been set. To do so, the same training set as the one used in reinforcement learning is used (Alien, Atlantis, Pooyan, Ms. Pac-Man). Four values for the exploration constant are tested: 0.01, 0.05, 0.1 and 0.2. Table 4.5 shows the results obtained for each game-value pair after 10 episodes.

To determine which exploration constant is the best to be used in the upcoming tests over the whole set of games, a similar methodology to the one applied in reinforcement learning is used. For each game, all the scores are divided by the maximum obtained score to get how good is an specific constant related to the best one. Then, the average of these results is computed for each constant; the average closest to 1 is considered to be the best (at least for this set of games).

\[
\text{avg}_{c=0.01} = \frac{2901}{2901} + \frac{153230}{153230} + \frac{12012.5}{12397.5} + \frac{4600}{5133} \approx 0.9663
\]

\[
\text{avg}_{c=0.05} = \frac{2624}{2901} + \frac{144830}{153230} + \frac{12385.5}{12397.5} + \frac{5133}{5133} \approx 0.9622
\]

<table>
<thead>
<tr>
<th>Game</th>
<th>α</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alien</td>
<td>0.01</td>
<td>2901</td>
</tr>
<tr>
<td></td>
<td>0.05</td>
<td>2624</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>2375</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>2328</td>
</tr>
<tr>
<td>Atlantis</td>
<td>0.01</td>
<td>153230</td>
</tr>
<tr>
<td></td>
<td>0.05</td>
<td>144830</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>145470</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>151820</td>
</tr>
<tr>
<td>Pooyan</td>
<td>0.01</td>
<td>12012.5</td>
</tr>
<tr>
<td></td>
<td>0.05</td>
<td>12385.5</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>12298</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>12397.5</td>
</tr>
<tr>
<td>Ms. Pac-Man</td>
<td>0.01</td>
<td>4600</td>
</tr>
<tr>
<td></td>
<td>0.05</td>
<td>5133</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>4983</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>4871</td>
</tr>
</tbody>
</table>

Table 4.5: Evaluation of different exploration constants $c$ in the UCT algorithm
\[
\text{avg}_{c=0.1} = \frac{2375 + 145470 + 12397.5 + 4983}{4} \approx 0.9327
\]

\[
\text{avg}_{c=0.2} = \frac{2328 + 151280 + 12397.5 + 4871}{4} \approx 0.9347
\]

As the results of the averages show, the differences between the values are not very different. In fact, games usually perform in a similar way independently of the value of the constant. Nevertheless, the lower values (0.01, 0.05) achieve better results than the higher ones (0.1, 0.2). Since 0.01 is the constant that gives the best results, it is the one used in the remaining games during the execution of UCT with the whole set of games.

**Experiments with UCT-RAVE**

To perform the tests with UCT using RAVE, the same set of games as before is used: Alien, Atlantis, Ms. Pacman and Pooyan. All the parameters are set with the same values than in standard UCT except for the exploration constant, which has been set to 0 according to previous successful experiments that used RAVE (e.g. in Go and Hex) [20].

Besides, it is necessary to give a value to the parameter to helps to determine the \( \alpha \) value at each node. To approximately know which values of the RAVE parameter we should use, it is convenient to get the maximum number of AMAF visits in any node. By knowing this amount of visits, we can set an approximate maximum value for the RAVE parameter. Table 4.6 shows the maximum number of AMAF visits received by a node during one episode in each game.

<table>
<thead>
<tr>
<th>Game</th>
<th>Maximum Number AMAF Visits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alien</td>
<td>1935</td>
</tr>
<tr>
<td>Atlantis</td>
<td>3428</td>
</tr>
<tr>
<td>Ms. Pac-Man</td>
<td>2790</td>
</tr>
<tr>
<td>Pooyan</td>
<td>3588</td>
</tr>
</tbody>
</table>

**Table 4.6:** Maximum number of AMAF visits in any node of the UCT tree for each training game
Given the results of table 4.6, some tests are performed using 6 different values of the parameter: 50, 100, 500, 1000, 2000 and 4000. In all cases they are executed for 10 episodes. Table 4.7 shows the scores for each game-value pair.

<table>
<thead>
<tr>
<th>Game</th>
<th>RAVE parameter</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alien</td>
<td>50</td>
<td>2552</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>2699</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>2509</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>2911</td>
</tr>
<tr>
<td></td>
<td>2000</td>
<td>2688</td>
</tr>
<tr>
<td></td>
<td>4000</td>
<td>2682</td>
</tr>
<tr>
<td>Atlantis</td>
<td>50</td>
<td>148930</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>147210</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>151260</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>148050</td>
</tr>
<tr>
<td></td>
<td>2000</td>
<td>149130</td>
</tr>
<tr>
<td></td>
<td>4000</td>
<td>149020</td>
</tr>
<tr>
<td>Ms. Pac-Man</td>
<td>50</td>
<td>4378</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>5141</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>4496</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>5608</td>
</tr>
<tr>
<td></td>
<td>2000</td>
<td>4710</td>
</tr>
<tr>
<td></td>
<td>4000</td>
<td>4643</td>
</tr>
<tr>
<td>Pooyan</td>
<td>50</td>
<td>12163</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>11802</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>12168</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>12498</td>
</tr>
<tr>
<td></td>
<td>2000</td>
<td>12127.5</td>
</tr>
<tr>
<td></td>
<td>4000</td>
<td>11818</td>
</tr>
</tbody>
</table>

Table 4.7: Evaluation of diverse values of the RAVE parameter in UCT-RAVE

From the results in table 4.7, we evaluate which value of the RAVE parameter usually performs better; to do that, we will follow the same methodology used when evaluating which UCT exploration constant gave the best results (remember that, in this case, the exploration constant has been set to 0). The RAVE parameter is named $r$ for convenience in the following equations:

$$\text{avg}_{r=50} = \frac{2552 + 148930 + 4378 + 12163}{4} \approx 0.9038$$
Evaluation

\[
\text{avg}_{r=100} = \frac{2699 + 147210 + 5141 + 11802}{4} \approx 0.9404
\]

\[
\text{avg}_{r=500} = \frac{2509 + 151260 + 4496 + 12168}{4} \approx 0.9093
\]

\[
\text{avg}_{r=1000} = \frac{2911 + 148050 + 5608 + 12498}{4} \approx 0.9947
\]

\[
\text{avg}_{r=2000} = \frac{2688 + 149130 + 4710 + 12127.5}{4} \approx 0.9299
\]

\[
\text{avg}_{r=4000} = \frac{2682 + 149020 + 4643 + 11818}{4} \approx 0.92
\]

From the previous equations, we observe that there is neither a huge difference between the values of the RAVE parameter nor a tendency between them (e.g. the performance does not decrease if the parameter increases). However, the best value of the parameter is 1000 (in three of the four tested games, it provides the best results).

Depending on the number of AMAF visits received by a node, we know if this parameter gives importance to the AMAF counter or not. Nevertheless, it is worth noting that this counter is important because the higher the RAVE parameter is, the more relevance the AMAF counter has; besides, the importance of this parameter highly depends on the game. For example, it was shown in table 4.6 that the maximum number of visits received by a node in Alien was 1935. Therefore, we can affirm that Alien gives more importance to the AMAF visits than the other games.

It is also interesting to compare the best results of UCT with and without using RAVE (independently of the exploration constant and the RAVE parameter). Table 4.8 summarizes the best results of these algorithms:
The average improvement of UCT with RAVE over UCT without it can be computed as follows:

\[
\text{improvement} = \frac{2911 - 2901}{4} + \frac{151260 - 153230}{4} + \frac{5608 - 5133}{4} + \frac{12498 - 12397.5}{4} \approx 1.0228
\]

From this result, we conclude that UCT-RAVE performs slightly better than standard UCT. Nevertheless, it would be convenient to execute much more episodes in some other games to really see if there is an improvement by using RAVE. Also remember that we have implemented a simplified version, so the performance might be a bit different from the real one.

### 4.1.2 Evaluation with the Full Set of Games

The algorithms used for individual learning are gradient-descent Sarsa with linear function approximation and UCT. Some of the parameters are shared by both algorithms, while others are particular to each one. Table 4.9 summarizes which were the parameters selected before from the training set (Alien, Atlantis, Ms. Pac-Man, Pooyan).

A change has been done respect to the parameters used before in Sarsa: when working with the training set, the maximum number of episodes was 20000, while now it is 30000 (as indicated in the table) since it was considered that better results could be achieved.

The number of episodes in UCT has also been changed from 10 to 5 because the scores do not vary too much by taking a lower quantity of episodes to compute the average. Besides, given the execution time of the previous games, we saw that each of them had different execution times per action, so episodes were longer in some games and shorter in others. Thus, to fit the temporal planning of the project, each game is executed for 5 episodes.

<table>
<thead>
<tr>
<th>Game</th>
<th>Best Stantard UCT Score</th>
<th>Best UCT-RAVE Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alien</td>
<td>2901</td>
<td>2911</td>
</tr>
<tr>
<td>Atlantis</td>
<td>153230</td>
<td>151260</td>
</tr>
<tr>
<td>Ms. Pac-Man</td>
<td>5133</td>
<td>5608</td>
</tr>
<tr>
<td>Pooyan</td>
<td>12397.5</td>
<td>12498</td>
</tr>
</tbody>
</table>

Table 4.8: Best results achieved with UCT with and without using RAVE
In the case of reinforcement learning (Sarsa), the functions learned at the end of the 30000th episode are saved. Then, these functions are loaded and the game is played for 1000 episodes without learning: greedy action selection is performed on the action-values obtained from the current feature vector and the frozen action functions. Moreover, the random initialization of the game should be enabled to evaluate the functions in different situations.

Table 4.10 shows the results obtained for 61 Atari games. For each game, the scores of Sarsa, UCT, a random agent and a human are shown.

The random agent picks an action randomly every 10 frames (in this case, the frame skip was also set to 10 for a more realistic comparison between algorithms). It constitutes a baseline for knowing how well Sarsa and UCT perform respect to the theoretically worst agent.

On the other hand, the human results allow us to know whether the used algorithms perform compared to humans. These results are partially obtained from Mnih et al. [16]: a professional games tester played 49 games for 20 episodes of a maximum of 5 minutes. The tester practiced for 2 hours with each game.
The remaining 12 games (denoted by an asterisk *) were played 5 times for a maximum of 5 minutes by the author of this thesis. The practice time was about 20 minutes for each game. However, the obtained scores might not be as high as the ones that a professional tester could get. The best results are emphasized in bold.

<table>
<thead>
<tr>
<th>Game</th>
<th>Sarsa</th>
<th>UCT</th>
<th>Random</th>
<th>Human</th>
</tr>
</thead>
<tbody>
<tr>
<td>Air Raid</td>
<td>4025</td>
<td>17010</td>
<td>556.45</td>
<td>1710*</td>
</tr>
<tr>
<td>Alien</td>
<td>593.29</td>
<td>2901</td>
<td>300.79</td>
<td>6875</td>
</tr>
<tr>
<td>Amidar</td>
<td>142.602</td>
<td>234.6</td>
<td>6.866</td>
<td>1676</td>
</tr>
<tr>
<td>Assault</td>
<td>292.425</td>
<td>1567.8</td>
<td>271.278</td>
<td>1496</td>
</tr>
<tr>
<td>Asterix</td>
<td>580.2</td>
<td>11060</td>
<td>276.25</td>
<td>8503</td>
</tr>
<tr>
<td>Asteroids</td>
<td>836.42</td>
<td>1186</td>
<td>634.25</td>
<td>13157</td>
</tr>
<tr>
<td>Atlantis</td>
<td>24667</td>
<td>153230</td>
<td>11034.9</td>
<td>29028</td>
</tr>
<tr>
<td>Bank Heist</td>
<td>140.33</td>
<td>58</td>
<td>13.48</td>
<td>734.4</td>
</tr>
<tr>
<td>Battle Zone</td>
<td>17458</td>
<td>78600</td>
<td>2447</td>
<td>37800</td>
</tr>
<tr>
<td>Beam Rider</td>
<td>1285.92</td>
<td>3786</td>
<td>440.764</td>
<td>5775</td>
</tr>
<tr>
<td>Berzerk</td>
<td>564.3</td>
<td>704</td>
<td>134.78</td>
<td>2190*</td>
</tr>
<tr>
<td>Bowling</td>
<td>53.773</td>
<td>23.6</td>
<td>24.645</td>
<td>154.8</td>
</tr>
<tr>
<td>Boxing(^1)</td>
<td>57.504</td>
<td>33</td>
<td>-2.201</td>
<td>4.3</td>
</tr>
<tr>
<td>Breakout</td>
<td>6</td>
<td>249.8</td>
<td>0.857</td>
<td>31.8</td>
</tr>
<tr>
<td>Carnival</td>
<td>2258.73</td>
<td>4258</td>
<td>620.32</td>
<td>2580*</td>
</tr>
<tr>
<td>Centipede</td>
<td>5793.79</td>
<td>25927</td>
<td>1717.5</td>
<td>11963</td>
</tr>
<tr>
<td>Chopper Command</td>
<td>962.5</td>
<td>840</td>
<td>715.2</td>
<td>9882</td>
</tr>
<tr>
<td>Crazy Climber</td>
<td>11300</td>
<td>1780</td>
<td>8297.3</td>
<td>35411</td>
</tr>
<tr>
<td>Defender</td>
<td>3797.9</td>
<td>2020</td>
<td>2403.6</td>
<td>13300*</td>
</tr>
<tr>
<td>Demon Attack</td>
<td>800.155</td>
<td>14655</td>
<td>124.12</td>
<td>3401</td>
</tr>
<tr>
<td>Double Dunk</td>
<td>-23.102</td>
<td>18.4</td>
<td>-20.252</td>
<td>-15.5</td>
</tr>
<tr>
<td>Elevator Action</td>
<td>180.4</td>
<td>4180</td>
<td>402.5</td>
<td>12520*</td>
</tr>
<tr>
<td>Enduro</td>
<td>134.594</td>
<td>140.8</td>
<td>0.173</td>
<td>309.6</td>
</tr>
</tbody>
</table>

\(^1\)Video that shows the learning process using Sarsa in Boxing: [https://youtu.be/98WewPvzagw](https://youtu.be/98WewPvzagw)
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fishing Derby</td>
<td>-99</td>
<td>-16.4</td>
<td>-90.58</td>
<td>5.5</td>
</tr>
<tr>
<td>Freeway</td>
<td>19.847</td>
<td>0.2</td>
<td>0.055</td>
<td>29.6</td>
</tr>
<tr>
<td>Frostbite</td>
<td>134.46</td>
<td>274</td>
<td>62.54</td>
<td>4335</td>
</tr>
<tr>
<td>Gopher</td>
<td>541.16</td>
<td><strong>9944</strong></td>
<td>224.32</td>
<td>2321</td>
</tr>
<tr>
<td>Gravitar</td>
<td>500</td>
<td>2270</td>
<td>176.8</td>
<td>2672</td>
</tr>
<tr>
<td>H.E.R.O</td>
<td>12025.6</td>
<td>8566</td>
<td>1496.92</td>
<td><strong>25763</strong></td>
</tr>
<tr>
<td>Ice Hockey</td>
<td>-7.599</td>
<td><strong>37.4</strong></td>
<td>-11.771</td>
<td>0.9</td>
</tr>
<tr>
<td>James Bond</td>
<td><strong>431.1</strong></td>
<td>200</td>
<td>32.65</td>
<td>406.7</td>
</tr>
<tr>
<td>Journey Escape</td>
<td>-29077.4</td>
<td>-720</td>
<td>-21619.2</td>
<td>0*</td>
</tr>
<tr>
<td>Kangaroo</td>
<td>600</td>
<td>1300</td>
<td>27</td>
<td><strong>3035</strong></td>
</tr>
<tr>
<td>Krull</td>
<td><strong>4427.38</strong></td>
<td>3622</td>
<td>1447.39</td>
<td>2395</td>
</tr>
<tr>
<td>Kung Fu Master</td>
<td>7842.6</td>
<td>8440</td>
<td>170</td>
<td><strong>22736</strong></td>
</tr>
<tr>
<td>Montezuma’s Revenge</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td><strong>4367</strong></td>
</tr>
<tr>
<td>Ms. Pac-Man</td>
<td>1372.82</td>
<td>4600</td>
<td>374.15</td>
<td><strong>15693</strong></td>
</tr>
<tr>
<td>Name This Game</td>
<td>2011.97</td>
<td><strong>7806</strong></td>
<td>1863.05</td>
<td>4076</td>
</tr>
<tr>
<td>Phoenix</td>
<td>943.31</td>
<td><strong>4944</strong></td>
<td>612.88</td>
<td>3262*</td>
</tr>
<tr>
<td>Pitfall</td>
<td>-24</td>
<td>-3</td>
<td>-222.789</td>
<td><strong>3068.6</strong>*</td>
</tr>
<tr>
<td>Pong</td>
<td>-21</td>
<td><strong>21</strong></td>
<td>-20.452</td>
<td>9.3</td>
</tr>
<tr>
<td>Pooyan</td>
<td>653.845</td>
<td><strong>12012.5</strong></td>
<td>274.95</td>
<td>572*</td>
</tr>
<tr>
<td>Private Eye</td>
<td>99.9</td>
<td>100</td>
<td>31.826</td>
<td><strong>69571</strong></td>
</tr>
<tr>
<td>Q*Bert</td>
<td>450</td>
<td>11985</td>
<td>206.75</td>
<td><strong>13455</strong></td>
</tr>
<tr>
<td>River Raid</td>
<td>1601.67</td>
<td>5552</td>
<td>820.3</td>
<td><strong>13513</strong></td>
</tr>
<tr>
<td>Road Runner</td>
<td>0</td>
<td><strong>31640</strong></td>
<td>4.7</td>
<td>7845</td>
</tr>
<tr>
<td>Robotank</td>
<td>30.896</td>
<td><strong>42.4</strong></td>
<td>2.653</td>
<td>11.9</td>
</tr>
<tr>
<td>Seaquest</td>
<td>606.76</td>
<td>590</td>
<td>57.88</td>
<td><strong>20182</strong></td>
</tr>
<tr>
<td>Skiing</td>
<td>-13726.5</td>
<td>-24978</td>
<td>-16770.2</td>
<td><strong>-3482.2</strong>*</td>
</tr>
<tr>
<td>Solaris</td>
<td>2166.08</td>
<td>1276</td>
<td>1651.58</td>
<td><strong>16076</strong>*</td>
</tr>
<tr>
<td>Space Invaders</td>
<td>446.18</td>
<td><strong>2646</strong></td>
<td>143.36</td>
<td>1652</td>
</tr>
<tr>
<td>Star Gunner</td>
<td>1000</td>
<td>160</td>
<td>594.8</td>
<td><strong>10250</strong></td>
</tr>
</tbody>
</table>
The results of table 4.10 are summarized in table 4.11. In almost half of the games (30), learning algorithms surpass human performance, while the other half (31) is completely dominated by humans.

Table 4.11: Summary of the times each agent (Sarsa, UCT, random and human) has obtained the highest score in a game

Besides, to rank which agent performs better, we can use the formula 4.1 (used in the International Planning Competition) to give a normalized score to each method in each game:

\[
\frac{1}{1 + \log_{10}\left(\frac{T^*}{T}\right)}
\]  

(4.1)

where \(T^*\) is the difference between the best and the worst score, and \(T\) is the difference between the score of the method we want to evaluate and the worst score. For example, if we wanted to evaluate Sarsa score of Air Raid, \(T^*\) would be the difference between the UCT score and the random score, whereas \(T\) would be the difference between the Sarsa
score and the random score:

\[ \frac{1}{1 + \log_{10} \left( \frac{s_{\text{uct}} - s_{\text{random}}}{s_{\text{sarsa}} - s_{\text{random}}} \right)} = \frac{1}{1 + \log_{10} \left( \frac{17010 - 556.45}{20230 - 556.45} \right)} \approx 0.5966 \]

Table 4.12 shows the normalized scores for all tested agents. Each score is the average of all normalized scores for each game.

<table>
<thead>
<tr>
<th>Agent</th>
<th>Normalized Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sarsa</td>
<td>0.4724</td>
</tr>
<tr>
<td>UCT</td>
<td>0.7166</td>
</tr>
<tr>
<td>Random</td>
<td>0.0973</td>
</tr>
<tr>
<td>Human</td>
<td>0.8438</td>
</tr>
</tbody>
</table>

Table 4.12: Normalized scores for Sarsa, UCT, random and human agents

As the table indicates, humans outperform all other agents and both learning algorithms greatly improve the results obtained by a random agent.

It is worth noting that UCT is much closer to humans than Sarsa (as we could deduce from table 4.11). Actually, UCT obtains higher scores than Sarsa in 45 of the 61 games, while Sarsa is better in 15 games (in the remaining game, Montezuma’s Revenge, they get the same score).

If the normalized score is computed now by comparing both learning algorithms (Sarsa and UCT) with the random and human agents, we can appreciate how far AI algorithms are from human performance. Table 4.13 shows that current learning algorithms are very close to perform as good as humans and overcome random agents without problems.

<table>
<thead>
<tr>
<th>Agent</th>
<th>Normalized Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sarsa &amp; UCT</td>
<td>0.8058</td>
</tr>
<tr>
<td>Random</td>
<td>0.0000</td>
</tr>
<tr>
<td>Human</td>
<td>0.8427</td>
</tr>
</tbody>
</table>

Table 4.13: Normalized scores for learning, random and human agents

Although UCT tends to perform well, there are some games with which its performance is almost as bad as the random. This behavior occurs because of the random simulations
during the default policy phase of the algorithm: these simulations do not allow to get scores which are difficult to reach. For example, if an action/movement must be executed for some frames to reach an score, then UCT will not work well because the random simulations do not allow to reach that score easily. Some examples in which UCT does not perform well are the following:

- **Freeway**: A chicken has to cross the street to get a point. The chicken can be knocked down by cars which drive in perpendicular direction. The animal goes some steps back if it collides with a car.

  During the random simulations of this game, the chicken should be capable of crossing the street to get a reward in order to guide the proper creation of the tree to get a high score. However, random behavior causes the chicken to go up and down continuously without crossing the street. Thus, in few occasions it will get an overall score higher than 0.

- **Montezuma’s Revenge**: It is the only game in the table which does not obtain score for any agent except for the human one. The player controls a character through distinct rooms in a kind of subterranean maze in search of treasures. To find these treasures, some keys have to be collected to open doors to other rooms.

  This game is much more difficult than Freeway in the sense that it does not only require to go up and down. To reach the first score in this game, the player has to go downstairs, move right, jump twice, go downstairs again, move left without crashing with an obstacle, go upstairs and jump. Besides, it is very easy to die in this game, so failing a single action in an unappropriated moment results in finishing the game quickly.

  Therefore, if random simulations are run, it is almost impossible to reach that reward since incorrect actions will continuously be used.

- **Ms. Pac-Man**: UCT works relatively well in this game. However, there are specific situations during the game in which it does not.
In general the problem is that if few sweets remain to be eaten and the character is far from them, then the random simulations will rarely allow the algorithm to set a path to those sweets. Consequently, the character will continuously oscillate. In section 3.3.2 a case in which the undesired behavior took place is explained.

**Pitfall:** In this game inspired by Indiana Jones, the character has to overcome different obstacles (e.g. crocodiles, barrels, snakes or holes) by jumping and moving left or right.

The game starts with an initial score of 2000, although our final score will be the increment or decrement respect to it. For example, if we get a score of 3000 at the end of the game, the real score is $3000 - 2000 = 1000$.

The base score (2000) is incremented if the character finds a treasure, or decremented if it collides with a barrel or falls in a finite hole (i.e. there are stairs to return to the upper floor).

As in the other games, reaching the treasure is quite difficult. The character must go through diverse stages until it finds the treasure which gives a huge reward. Thus, lots of frames should be simulated during the default policy to start guiding the tree towards the treasure.

**Venture:** The game starts in a big dungeon where there are several rooms in which the controlled character (a pink point) can enter, and there are monsters which cannot be defeated.

As soon as the character enters in a room, the image is zoomed and some monsters appear; the enemies of these rooms can be killed by shooting a bullet to them (although any score is obtained). To get score, it is necessary to collect the treasures of each room.

To get some score in UCT is difficult because the character firstly needs to enter into a room with very narrow doors, and then navigate the room to reach the treasure. Therefore, many frames are required to be explored in order to get score. Again
UCT starts to oscillate since it does not find a path towards reward and even does not enter in any of the rooms.

A possible workaround to this problem would consist in increasing the number of frames simulated during the default policy with the hope that a reward is eventually reached. Also it could be possible to achieve better results by increasing the number of iterations per node to get a bigger tree. However, as we have seen before, increasing these parameters result in a higher execution time (which can be exponential, see section 2.5.3). Random behavior during the default policy simulations could be avoided by using an heuristic that lead us to a reward easily; however, it is probably difficult to formulate an heuristic which generalizes to all Atari games given their different nature.

![Figure 4.4: Screenshots of Pitfall, Montezuma’s Revenge and Freeway](image)

Sarsa suffers from the same problem as UCT; however, in some cases like Freeway and Venture, it finally manages to score points. In the beginning, Sarsa explores different actions continuously (almost it behaves as a random agent) until it reaches the first reward. Unlike UCT, Sarsa saves a model with information about the states and actions applied; therefore, it uses this data to choose actions which have proved to be useful in the past. Thus, when rewards are started to be obtained the way in which the agent behaves changes to get more reward by using the same actions as before, while UCT does not\(^2\).

Figure 4.5 shows the previously described behavior in Venture and Freeway. In this figure and upcoming figures, the random and human curves are constant, whereas the Sarsa curve represents the learning process during 30000 episodes (each point is the average of the last 1000 episodes). In UCT, since far fewer episodes are executed, the curve is not

\(^2\)Video of Sarsa and UCT performances in Freeway and Venture: [https://youtu.be/qw-93T-vUsw](https://youtu.be/qw-93T-vUsw)
soft and is adapted to the length of Sarsa (30000) to provide a proper comparison; besides, each point is the average of all episodes until that moment.

In Venture, for 1530 episodes it does not receive any reward greater than 0, and it is not until the 5000th episode when it starts to grow very quickly. Something similar occurs with Freeway (although it gets rewards greater than 0 in just 4 episodes).

![Figure 4.5: Results of Freeway and Venture when learning alone](image)

Although we have seen cases in which UCT does not work as expected, it usually outperforms humans and other algorithms (see figure 4.6). For example, Pong and Breakout are two games in which UCT is specially strong since a small lookup of frames is required to detect possible actions to get rewards. On the other hand, Sarsa does not perform well in these games: it has the same problem as before but it does not find a way to improve since rewards greater than zero are not being obtained\(^3\).

<table>
<thead>
<tr>
<th></th>
<th>Freeway</th>
<th>Venture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sarsa</td>
<td><img src="image" alt="Sarsa" /></td>
<td><img src="image" alt="Sarsa" /></td>
</tr>
<tr>
<td>UCT</td>
<td><img src="image" alt="UCT" /></td>
<td><img src="image" alt="UCT" /></td>
</tr>
<tr>
<td>Human</td>
<td><img src="image" alt="Human" /></td>
<td><img src="image" alt="Human" /></td>
</tr>
<tr>
<td>Random</td>
<td><img src="image" alt="Random" /></td>
<td><img src="image" alt="Random" /></td>
</tr>
</tbody>
</table>

There are also games in which neither UCT nor Sarsa perform well. As previously explained, UCT performs poorly in Montezuma’s Revenge and Pitfall. The same occurs with Sarsa for the same reasons: the reward is very difficult to get (as in Pong and Breakout).

\(^3\)Video of Sarsa and UCT performances in Breakout and Pong: [https://youtu.be/axkIkJAy0](https://youtu.be/axkIkJAy0)
Figure 4.6: Results of Breakout and Pong when learning alone

All Sarsa actions will have the same value until a reward is reached, so the behavior will be similar to the random agent\(^4\).

Apart from the previous cases, there are others which exhibit different behaviors (see figure 4.7).

In the case of Atlantis, UCT is the best agent with a huge difference respect to the other ones. Although Sarsa almost surpasses human performance in Atlantis, its average score decreased during the last 25000 episodes without rising again\(^5\).

Krull exhibits a different behavior than Atlantis. The agent using Sarsa learns how to play this game incrementally until it outperforms UCT and humans. Besides, UCT also gets a higher scores than humans.

In Ms. Pac-Man, humans greatly outperform both UCT and Sarsa; moreover, there is also a notable difference of performance between UCT and Sarsa. However, it seems that

\(^4\)Video of Sarsa and UCT performances in Montezuma’s Revenge and Pitfall \url{https://youtu.be/j8weqY4G_4}

\(^5\)Video of Sarsa and UCT performances in Atlantis: \url{https://youtu.be/WCJ40X0CETQ}
eventually Ms. Pac-Man will be able to reach UCT because it is slowly learning to play better.

4.2 Learning in Multiple Games

In this section the results when multiple game learning are shown and compared to the ones obtained when learning individually. Concretely, these will be compared to the ones obtained in Sarsa because, as explained in the previous chapter, the algorithm used for this kind of learning is Sarsa as well.

4.2.1 Parameter Selection

The previously found general and Sarsa parameters in the previous section are used again. By employing the same values of these parameters, we ensure a proper comparison between the results of both types of learning. Table 4.9 summarized the previous parameters used.
in Sarsa (see All and Sarsa rows). Besides, there are some other parameters to be set that are particular to this kind of learning (see section 3.4).

Firstly, we have to decide the kind of learning steps that will be performed. Since performing action steps has the problem that some games might not entirely execute their episodes, the other approach (episode steps) is used. In this case, since each game is able to execute its episode without interruption, the previous problem does not arise. However, the problem of different episode lengths still exists: a game whose episodes are longer and that obtains positive reward more frequently might have more influence on the function being created. Although this problem happens, it is considered that this way can give better results because at least all episodes are fully executed in each game.

Besides, if actions steps were used, the features and actions used at each step should be defined; that is, whether we keep the features and actions separately for each game or we use them without caring from which game they are. Thus, since it is difficult to define which alternative is the best, it is another reason for which episode steps might be better.

Regarding the order in which games are chosen during the learning phase, since we want all games to execute the same number of episodes (remember we have chosen episode steps), they must be selected alternatively one after another. Therefore, the methods to be used are either ascending or descending order. The finally chosen method is ascending order.

The number of games used in each learning process is also important. The decision is to employ two games, although the code is prepared to support more. Two games will allow us to observe whether some game, $g_1$, particularly helps another game, $g_2$, to build a model that results in an increased score respect to the single game learning.

Also remember that the rewards have to be scaled to $\{-1, 0, 1\}$ depending on the sign of the original reward as specified in section 3.4.
4.2.2 Evaluation with a Partial Set of Games

To evaluate the process of learning in multiple games, we need a subset of games to perform the experiments. Since using the full set of games results in a huge amount of experiments to be run, we have selected 10 games (including those previously used in the training set in single game learning): Alien, Atlantis, Bank Heist, Beam Rider, Breakout, Demon Attack, Ms. Pac-Man, Pooyan, River Raid and Space Invaders. Thus, since we use 10 games, \(\frac{n(n-1)}{2} = \frac{10(10-1)}{2} = 45\) simulations are executed.

Table 4.14 shows the results for this set of simulations. It is read as follows: the cell \((i,j)\) (\(i\) is a row, \(j\) is a column) is the score obtained for game \(i\) after being trained with game \(j\). Cells where \(i = j\) contain the score of game \(i\) shown in table 4.10 (Sarsa column), and the highest scores are emphasized in bold in each row.

<table>
<thead>
<tr>
<th>Game</th>
<th>Alien</th>
<th>Atlantis</th>
<th>Bank Heist</th>
<th>Beam Rider</th>
<th>Breakout</th>
<th>Demon Attack</th>
<th>Ms. Pac-Man</th>
<th>Pooyan</th>
<th>River Raid</th>
<th>Space Invaders</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alien</td>
<td>593.29</td>
<td>1232.29</td>
<td>1711.78</td>
<td>714.06</td>
<td>596.54</td>
<td>733.83</td>
<td>582.55</td>
<td>718.73</td>
<td>894.28</td>
<td>574.73</td>
</tr>
<tr>
<td>Atlantis</td>
<td>1404.38</td>
<td>2458.38</td>
<td>3099.14</td>
<td>2689.62</td>
<td>2599.48</td>
<td>2731.11</td>
<td>1933.65</td>
<td>2138.07</td>
<td>2431.42</td>
<td>2215.33</td>
</tr>
<tr>
<td>Bank Heist</td>
<td>105.44</td>
<td>32623.5</td>
<td>24667</td>
<td>1111.27</td>
<td>714.06</td>
<td>596.54</td>
<td>733.83</td>
<td>582.55</td>
<td>718.73</td>
<td>894.28</td>
</tr>
<tr>
<td>Beam Rider</td>
<td>1058.59</td>
<td>638.48</td>
<td>616</td>
<td>1285.92</td>
<td>975.74</td>
<td>536.48</td>
<td>660</td>
<td>756</td>
<td>708.48</td>
<td>732.48</td>
</tr>
<tr>
<td>Breakout</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Demon Attack</td>
<td>521.38</td>
<td>701.34</td>
<td>785.34</td>
<td>895.41</td>
<td>800.16</td>
<td>245.45</td>
<td>880.82</td>
<td>812.06</td>
<td>682.51</td>
<td>1138.66</td>
</tr>
<tr>
<td>Ms. Pac-Man</td>
<td>900.9</td>
<td>1407.35</td>
<td>1273.91</td>
<td>635.68</td>
<td>1671.61</td>
<td>894.28</td>
<td>818.75</td>
<td>2147.07</td>
<td>812.06</td>
<td>1138.66</td>
</tr>
<tr>
<td>Pooyan</td>
<td>1138.15</td>
<td>859.95</td>
<td>413.54</td>
<td>1321.41</td>
<td>361.89</td>
<td>271.40</td>
<td>384.12</td>
<td>653.94</td>
<td>563.75</td>
<td>1449.21</td>
</tr>
<tr>
<td>River Raid</td>
<td>932.53</td>
<td>1095.35</td>
<td>935.11</td>
<td>1104.38</td>
<td>923.14</td>
<td>268.79</td>
<td>340.15</td>
<td>653.94</td>
<td>2147.07</td>
<td>1138.66</td>
</tr>
<tr>
<td>Space Invaders</td>
<td>288.24</td>
<td>364.98</td>
<td>206.9</td>
<td>368.35</td>
<td>224.59</td>
<td>176.85</td>
<td>239.64</td>
<td>345.76</td>
<td>446.18</td>
<td></td>
</tr>
</tbody>
</table>

**Table 4.14:** Scores for learning in multiple games experiments (scores in diagonal are the ones in table 4.10)

Table 4.15 contains all the scores of table 4.14 divided by the score obtained when learning in a single game (those scores in the diagonal). Thus, each cell can be understood as the improvement over the single game learning using Sarsa. If a cell \((i,j)\) is greater than 1, it means that game \(i\) performs better playing with game \(j\) rather than playing alone.

<table>
<thead>
<tr>
<th>Game</th>
<th>Alien</th>
<th>Atlantis</th>
<th>Bank Heist</th>
<th>Beam Rider</th>
<th>Breakout</th>
<th>Demon Attack</th>
<th>Ms. Pac-Man</th>
<th>Pooyan</th>
<th>River Raid</th>
<th>Space Invaders</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alien</td>
<td>1.000</td>
<td>2.077</td>
<td>1.971</td>
<td>1.204</td>
<td>1.096</td>
<td>0.922</td>
<td>1.211</td>
<td>1.801</td>
<td>0.968</td>
<td></td>
</tr>
<tr>
<td>Atlantis</td>
<td>1.323</td>
<td>1.000</td>
<td>1.209</td>
<td>1.134</td>
<td>1.093</td>
<td>0.974</td>
<td>1.191</td>
<td>0.910</td>
<td>0.981</td>
<td></td>
</tr>
<tr>
<td>Bank Heist</td>
<td>1.178</td>
<td>1.000</td>
<td>3.065</td>
<td>0.846</td>
<td>0.338</td>
<td>0.174</td>
<td>1.214</td>
<td>0.915</td>
<td>0.511</td>
<td></td>
</tr>
<tr>
<td>Beam Rider</td>
<td>0.824</td>
<td>0.479</td>
<td>0.737</td>
<td>1.000</td>
<td>0.979</td>
<td>0.427</td>
<td>0.544</td>
<td>0.858</td>
<td>0.594</td>
<td></td>
</tr>
<tr>
<td>Breakout</td>
<td>1.119</td>
<td>0.503</td>
<td>1.197</td>
<td>0.944</td>
<td>1.058</td>
<td>0.441</td>
<td>0.584</td>
<td>0.808</td>
<td>0.586</td>
<td></td>
</tr>
<tr>
<td>Demon Attack</td>
<td>0.652</td>
<td>0.901</td>
<td>0.409</td>
<td>0.665</td>
<td>1.124</td>
<td>0.314</td>
<td>0.725</td>
<td>1.905</td>
<td>0.511</td>
<td></td>
</tr>
<tr>
<td>Ms. Pac-Man</td>
<td>0.606</td>
<td>1.085</td>
<td>0.928</td>
<td>0.464</td>
<td>1.218</td>
<td>0.640</td>
<td>1.000</td>
<td>0.669</td>
<td>0.826</td>
<td></td>
</tr>
<tr>
<td>Pooyan</td>
<td>1.720</td>
<td>1.370</td>
<td>0.634</td>
<td>2.030</td>
<td>0.592</td>
<td>0.415</td>
<td>0.588</td>
<td>1.000</td>
<td>2.216</td>
<td>1.385</td>
</tr>
<tr>
<td>River Raid</td>
<td>0.943</td>
<td>0.885</td>
<td>0.414</td>
<td>0.724</td>
<td>0.629</td>
<td>0.314</td>
<td>0.625</td>
<td>1.000</td>
<td>2.216</td>
<td>1.385</td>
</tr>
<tr>
<td>Space Invaders</td>
<td>0.646</td>
<td>1.042</td>
<td>1.839</td>
<td>0.450</td>
<td>0.808</td>
<td>0.612</td>
<td>0.394</td>
<td>0.747</td>
<td>1.000</td>
<td></td>
</tr>
</tbody>
</table>

**Table 4.15:** Improvement index for each game combination respect to playing alone
From table 4.15, it is easy to know which games usually improve when playing with other games, and which games cause other games to improve their performance. In the next subsections these results are analyzed.

Performance Improvement

To know the average performance improvement for each game when playing with other games, the mean has to be computed for each row excluding the cell in the diagonal. Table 4.16 shows the average improvement when playing with another game; furthermore, the highest improvement (also without taking into account the scores in the diagonal), and the game with which it has been achieved are also written.

<table>
<thead>
<tr>
<th>Game</th>
<th>Average Improvement</th>
<th>Highest Improvement</th>
<th>Best Companion Game</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alien</td>
<td>1.3405</td>
<td>2.077</td>
<td>Atlantis</td>
</tr>
<tr>
<td>Atlantis</td>
<td>1.0069</td>
<td>1.323</td>
<td>Alien</td>
</tr>
<tr>
<td>Bank Heist</td>
<td>1.2058</td>
<td>3.065</td>
<td>Beam Rider</td>
</tr>
<tr>
<td>Beam Rider</td>
<td>0.5773</td>
<td>0.823</td>
<td>Alien</td>
</tr>
<tr>
<td>Breakout</td>
<td>0.4444</td>
<td>1.167</td>
<td>Bank Heist</td>
</tr>
<tr>
<td>Demon Attack</td>
<td>0.7866</td>
<td>1.123</td>
<td>Breakout</td>
</tr>
<tr>
<td>Ms. Pac-Man</td>
<td>0.7728</td>
<td>1.218</td>
<td>Breakout</td>
</tr>
<tr>
<td>Pooyan</td>
<td>1.2138</td>
<td>2.216</td>
<td>River Raid</td>
</tr>
<tr>
<td>River Raid</td>
<td>0.6108</td>
<td>0.841</td>
<td>Pooyan</td>
</tr>
<tr>
<td>Space Invaders</td>
<td>0.6782</td>
<td>1.042</td>
<td>Atlantis</td>
</tr>
<tr>
<td>Average</td>
<td>0.8637</td>
<td>1.489</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.16: Average and highest improvements of each game when playing with another game

From this table (4.16), it can be observed that 4 of the games generally improve when playing with other games: Alien, Atlantis, Bank Heist and Pooyan. On the other hand, there are 6 games which tend to achieve lower scores than when playing alone. The worst case is Breakout, whose average performance is less than half the score got when learning alone. The average improvement of all games (0.8637) indicates that games do not usually improve when playing with other games.

Nevertheless, there are just 2 games which do not improve its performance with any game: Beam Rider and River Raid. The other 8 games improve with some game and even they double (Alien, Pooyan) or triple (Bank Heist) their score respect to learning individually.
Actually, if the average improvement when playing with the best companion game is computed, it is greater than 1 (1.489). Therefore, it is clear that to achieve higher scores, the companion game matters. For example, if Breakout plays with Bank Heist it gets an score 7, while the score becomes 0 when it plays with River Raid.

An interesting fact is that although each game plays half of the episodes alone, its performance usually increases if an appropriate companion game is selected (i.e. the best case). Remember that each game plays alternatively an episode until an overall of 30000 episodes is reached; therefore, each game plays 15000 episodes. Thus, the number of contributions to the model/function are not as many as before, but the obtained score is usually higher.

Besides, the improvement for a pair of games can be reciprocal or unilateral. In the first case, both games involved in the learning process obtain higher scores than playing alone (e.g. the Alien - Atlantis pair). On the other hand, unilateral improvement occurs when just one of the two games achieves a higher score, while the other does not (e.g. the Breakout - Demon Attack pair). There are also cases in which the combined learning does not result in better results for any game respect to playing alone (e.g. the Beam Rider - River Raid pair). Reciprocal improvement takes place just 4 times: Alien - Atlantis, Alien - Bank Heist, Alien - Pooyan, and Atlantis - Bank Heist. On the other hand, unilateral improvement is given 21 times, while playing with another game does not result in better performance 20 times.

Learning in multiple games could result in an improvement over human performance. Table 4.17 shows the score obtained playing alone (using Sarsa), the best score playing with another game (see table 4.14), and the score obtained by a human. The best score for each game is emphasized in bold.

Although there are specific pairs of games which give better results than when playing individually, in most of the cases the improvement is not enough to outperform humans. However, in the case of the game Atlantis, the score obtained by humans is surpassed.

---

6Video showing the performance of Alien and Atlantis after learning alone and together: https://youtu.be/hwbUWZEUq8Q
when it plays with Alien. Besides, Pooyan playing with River Raid also performs better than humans; however, when Pooyan played alone it also beat humans.

<table>
<thead>
<tr>
<th>Game</th>
<th>Single Game Score</th>
<th>Best Multigame Score</th>
<th>Human</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alien</td>
<td>593.29</td>
<td>1232.29</td>
<td>6875</td>
</tr>
<tr>
<td>Atlantis</td>
<td>24667</td>
<td>32623.5</td>
<td>29028</td>
</tr>
<tr>
<td>Bank Heist</td>
<td>140.33</td>
<td>430.06</td>
<td>734.4</td>
</tr>
<tr>
<td>Beam Rider</td>
<td>1285.92</td>
<td>1058.59</td>
<td>5775</td>
</tr>
<tr>
<td>Breakout</td>
<td>6</td>
<td>7</td>
<td>31.8</td>
</tr>
<tr>
<td>Demon Attack</td>
<td>800.155</td>
<td>898.41</td>
<td>3401</td>
</tr>
<tr>
<td>Ms. Pac-Man</td>
<td>1372.82</td>
<td>1671.61</td>
<td>15693</td>
</tr>
<tr>
<td>Pooyan</td>
<td>653.845</td>
<td>1449.21</td>
<td>572</td>
</tr>
<tr>
<td>River Raid</td>
<td>1601.67</td>
<td>1346.78</td>
<td>13513</td>
</tr>
<tr>
<td>Space Invaders</td>
<td>446.18</td>
<td>464.98</td>
<td>1652</td>
</tr>
</tbody>
</table>

Table 4.17: Scores obtained through single and multiple game learning compared to the ones scored by humans. The set of games is the one used in learning in multiple games.

It is worth noting that a generalization of two similar graphical games (e.g. Space Invaders and Demon Attack) does not imply that their scores will increase by learning together in our implementation. Remember that our features are got from the RAM memory, so having similar graphics does not mean that they will have analogous memory organization; in fact, the use of the RAM memory depends on the programmer/s who made the game. This evidence is proven by table 4.16: games without any similarity in graphics, help each other to improve their scores (e.g. Alien and Atlantis).

The fact that some games perform better or worse when playing with other games might be due to exploration issues. Exploration might be either positive or negative. A game $g_1$ can encourage another game $g_2$ to explore new actions by giving high values to the parameters that cause these actions to be selected. Thus, if $g_2$ does not usually select them it is good for it so that the effect of this action in the game can be evaluated. Nevertheless, if exploration is excessively encouraged by $g_1$, then bad results will be obtained in $g_2$ if the exploratory action is actually harmful.

The similarity between the RAM memories of two games has consequences during the learning process. If two games do not have similar memories, then it is possible that each one will learn on its own independently of the other game. For example, if the first 512
bits of a game $g_1$ are always 0, and the last 512 bits of a game $g_2$ are always 0, then the parameter vector for each action will never mix their values, so they are learning independently. Thus, in this case, the exploration issues cannot take place.

On the contrary, when two games have many RAM bits in common, they get similar feature vectors. Hence, their action-values for a specific action will also be analogous, so exploration can be easily encouraged in one of the games. Note that the game (e.g. $g_1$) that might explore less is the one which obtains more normalized score during the episode; the reason behind this behavior is that this game will have much more influence on the model, so it will tend to act greedily while the other game, $g_2$, acts depending on the greedy actions of $g_1$ which can be exploratory for itself.

In general, as games do not tend to have totally independent RAM memories, it can be affirmed that playing with other games helps to define nuances in the learning process of each game through exploration. However, in some cases exploration can lead to undesired behavior that causes a decrease in the score.

**Effect on Improvement in Other Games**

Since we have seen that playing with any game does not always result in better performance, it is interesting to see which games cause another game to achieve higher scores. To do that, we compute the average of each column of table 4.15 without taking into account the diagonal values. Table 4.18 shows the average and best improvement caused by each game along with the affected game (i.e. the game that undergoes these improvements).

Table 4.18 shows that there are games that tend to help more than others in achieving higher scores. Concretely, these games are: Atlantis, Beam Rider and River Raid.

However, it seems that all of them (except for Ms. Pac-Man) often help another game during learning (see the column which contains the best relative scores). In fact, each game almost doubles (1.7 times) the score of another game on average.
It is worth noting that the games which are helped to improve are almost the same ones that obtained highest scores in table 4.16 (Alien, Bank Heist and Pooyan). Moreover, although River Raid and Beam Rider are games that help to increase the score of other games on average, they usually do not perform well when playing with other games (as shown in table 4.16).

As explained before, the similarity between RAM memories can be the main responsible for this behavior. If a game makes another to explore excessively, the performance of the second game will not be good. On the other hand, if exploration is sufficiently encouraged, then the other game might improve.
Chapter 5

Related Work

5.1 Deep Q-Networks (DQN)

Mnih et al. [16] have developed a new machine learning method which combines reinforcement learning with a type of artificial neural networks called deep neural networks. Concretely, they use deep convolutional neural networks, where hierarchical layers of tiled convolutional filters are used.

The reinforcement learning process using non-linear function approximation is known to be unstable. The causes of these problems are: the correlations in the sequence of observations, the small updates to the action-values $Q$, and the correlations between the current values of $Q$ and the target values $r + \gamma \max_{a'} Q(s', a')$.

Therefore, the authors had to tune the Q-learning algorithm to solve the problems which caused this undesired behavior. To remove the problems with correlations, they used a mechanism called experience replay that randomizes over the data. Besides, they used an iterative update that adjusts the action-values to the target values, which are periodically updated, to reduce the correlations between them.

They tested this method with Atari games by using the pixels and the score as input. The images were firstly preprocessed to reduce their dimensionality and to remove artifacts
of the Stella emulator. Remember that our approach used the RAM memory to build the features for the reinforcement learning algorithm (gradient-descent Sarsa), while the features used in DQN are obtained from the screen. As humans learn to play these games by watching the screen, their comparison between the algorithm and humans is fairer than ours since the learning process is done in equal conditions.

A set of 49 games was used to perform the experiments, which are then contrasted with scores obtained by a professional games tester. DQN outperformed humans in 23 out of the 49 games. Besides, their method also surpasses linear function approximation methods (e.g. gradient-descent Sarsa) in 43 games.

There are particular games in which the improvement respect to linear methods is significant. For example, Sarsa obtains very low scores in Pong and Breakout (-21 and 7 respectively in our experiments), which are games in which high precision is required to get reward (it is essential to touch the ball to keep playing); on the other hand, DQN gets 18.9 points in Pong and 401.2 in Breakout\(^1\). There is another pair of games which perform very well with the DQN: Road Runner and Crazy Climber have obtained 0 and 11300 points respectively in our Sarsa implementation, while in DQN they have scored 18257 and 114103 points respectively.

In shooting games, the scores increase in DQN respect to linear methods. For example, Space Invaders in our implementation obtained 446.18 points, while in DQN it scored 1976\(^2\). Maze games also get higher scores (e.g. Alien gets 593.29 with Sarsa and 3068 with DQN). However, there are games in which it performs poorly such as Montezuma’s Revenge (as we have seen in chapter 4, it is a challenging game to play).

Given the results of the paper, it can be affirmed that DQN is a promising technique for achieving human-level performance in domains with visual input and complicated interaction.

\(^1\)Video of DQN in Breakout: [http://goo.gl/zbP0bJ](http://goo.gl/zbP0bJ)

\(^2\)Video of DQN in Space Invaders: [http://goo.gl/Xnc1rz](http://goo.gl/Xnc1rz)
5.2 Classical Planning

Lipovetzky and Geffner have recently applied classical planning techniques in the Atari 2600 games [31]. They state that the action selection problem, where state transitions and rewards are unknown, can be understood as a shortest path problem. The search process in this case is blind because of the unknown transitions and rewards.

To deal with this problem they propose to use a kind of planning algorithm that combines blind search methods with the performance of classical planners. This algorithm is called Iterated Width (IW), which consists in series of calls IW(1), IW(2), . . . , IW(k), where IW(i) is a breadth-first search where states are pruned when they are not the first states to make true i atoms. This last condition is known as the novelty of the state.

The RAM memory of the console is used to represent the atoms. Remember that this memory is formed by 128 bytes. In their approach, unlike ours for reinforcement learning, they directly use bytes instead of decomposing them in bits. Each byte is associated with a variable $X_i$, $i = 1, \ldots, 128$ and has 256 possible values $x_j$. An state $s$ makes an atom $X_i = x_j$ true when the value of the $i^{th}$ byte in the RAM is $x_j$. Thus, accumulated reward is not considered for selecting states.

IW takes the planning problem as the input, while the output is the sequence of actions needed to solve the problem. The problem is considered to be solved when either the goal or the maximum number of frames are reached. The best path is the one that gives the highest accumulated reward during the game.

The authors also consider another algorithm called 2BFS, in which two state queues are maintained: one ordered by novelty and the other by accumulated reward. Then, in one iteration the algorithm picks the first state from one queue, and in the next iteration from the other.

The results obtained by IW and 2BFS in 54 different games are compared to the ones got by UCT and BFS. IW gets the highest scores 26 times, while 2BFS does in 13 cases. Besides, BFS just performs once as the best algorithm, and UCT 19 times. Thus, it is clear
that classical planning algorithms achieve higher scores than state of the art algorithms used in Atari games\(^3\).

\(^3\)Videos of the performance of IW and 2BFS in Freeway, Breakout and Space Invaders: https://youtu.be/0eDxyXn2pKo?list=PLXpQcXUQ_CwenUazUivhXyYvjuS6KQ0I0
Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis we have tested machine learning algorithms in Atari games. Specifically, two kinds of methods have been used: reinforcement learning and search methods. The reinforcement learning algorithm selected was gradient-descent Sarsa (which uses function approximation), while UCT was used for search.

Firstly, we tested both methods to learn in a single game. An small set of games was used to find the best possible parameters to perform an evaluation; from this parameter search, we saw that both algorithms are very sensible to value changes, i.e. their performance may greatly change depending on the values of these parameters. Besides, since there was not a value that worked equally well for all games, it is clear that each game has its own optimum values for parameters.

Then we compared these results to the ones achieved by humans. Humans outperformed machine learning techniques by a little margin, although the search algorithm proved to be much closer to human performance than the one using reinforcement learning. A possible cause for which this occurs can be that the behavior of Atari games cannot be linearly
approximated. Moreover, both approaches do not work well in those games in which is
difficult to get reward (i.e. large number of frames until a reward is obtained).

Secondly, the reinforcement learning approach was used to learn from multiple games
simultaneously. The results showed that better performances are not always achieved
when a generalization is made between two games.

However, in most of the cases, each game had a companion game with which it achieved
better performance than playing alone. Besides, half of the overall of the episodes were
executed for each game, so it was interesting that despite having less "opportunities" to
influence the model, the results were better.

The process of learning from multiple games could result in reciprocal (both games im-
prove), unilateral (one game improves) or no improvement. Besides, it has been found
that there are games that help other games to increase their scores.

We think that during the process of learning, each game encouraged the other game to
explore its set of actions by giving high values to parameters which were barely considered
before. If exploration is excessively encouraged, the agent will never find the greedy actions
to execute in order to obtain reward; on the other hand, if sufficient exploration is done,
the agent will be able to differentiate the greedy actions from the non-greedy actions.

Furthermore, it is possible that two games with distinct memory organizations barely
affect each other. Thus, the learning process is done just like if each game played alone
with little influence from the other game in some areas of the parameter vector.

\section{Future Work}

In our reinforcement learning approach we used the RAM memory of the Atari 2600 to
build the feature vectors. Then, the results were compared to the ones obtained by a
human. However, a human does not learn to play by looking at the memory bits, but by
watching an screen. Thus, it would be interesting to use image processing (e.g. to detect
Conclusions and Future Work

objects) instead of the RAM memory in order to perform a comparison in more similar conditions.

As explained in chapter 5, the experiments with the DQN algorithm have already used images as an input to learning. Besides, the IW and 2BFS algorithms, also presented in that chapter, use RAM data as our approach. Therefore, it would be also interesting to test these algorithms with data from the screen.

Image processing could also be interesting to use while learning from multiple games simultaneously. This time, however, games which are graphically different should give bad results, while those which are similar should return good results. An interesting experiment to perform would be to train from two graphically similar games (e.g. Beam Rider and Demon Attack), and then testing the model in a third similar game (e.g. Phoenix). By doing this we could evaluate how well generalizations are made from analogous games.

Another possible experiment would consist in using deep neural networks to learn from multiple games. DQN has proved to be a good algorithm in the Atari games, so better results could be probably achieved by using it with features extracted from either the RAM or the screen.

There exist methods that combine reinforcement learning with search. Dyna-2 [32] is an algorithm of such type and has been a successful method in Computer Go by mixing Sarsa and UCT.

Respect to the UCT algorithm, enhancements would be interesting to be done, especially in the default policy. As we have seen, the random behavior during the simulations might not result in good performance; thus, formulating heuristics that worked in the entire set of games is a major challenge to overcome.
Appendix A

Sarsa Training Results

Table A.1 shows the average scores obtained for training and evaluation. All games were trained for 30000 episodes and evaluated with the resulting function during 1000 episodes (no learning involved).

The average scores for the last 1000 training episodes and for the whole set of episodes (30000) are presented. The average scores during the 1000 episodes are also shown.

<table>
<thead>
<tr>
<th>Game</th>
<th>Training (1000)</th>
<th>Training (30000)</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Air Raid</td>
<td>2863.675</td>
<td>2129.89</td>
<td>4025</td>
</tr>
<tr>
<td>Alien</td>
<td>844</td>
<td>1019.86</td>
<td>593.29</td>
</tr>
<tr>
<td>Amidar</td>
<td>158.339</td>
<td>112.256</td>
<td>142.602</td>
</tr>
<tr>
<td>Assault</td>
<td>302.442</td>
<td>295.59</td>
<td>292.425</td>
</tr>
<tr>
<td>Asterix</td>
<td>1226.4</td>
<td>1055.48</td>
<td>580.2</td>
</tr>
<tr>
<td>Asteroids</td>
<td>929.21</td>
<td>863.441</td>
<td>836.42</td>
</tr>
<tr>
<td>Atlantis</td>
<td>25160.3</td>
<td>26125.2</td>
<td>24667</td>
</tr>
<tr>
<td>Bank Heist</td>
<td>271.68</td>
<td>238.814</td>
<td>140.33</td>
</tr>
<tr>
<td>Battle Zone</td>
<td>17094</td>
<td>15900.6</td>
<td>17458</td>
</tr>
<tr>
<td>Beam Rider</td>
<td>970.984</td>
<td>898.093</td>
<td>1285.92</td>
</tr>
<tr>
<td>Berzerk</td>
<td>518.34</td>
<td>471.716</td>
<td>564.3</td>
</tr>
<tr>
<td>Game</td>
<td>Sarsa</td>
<td>Sarsa (1)</td>
<td>Sarsa (2)</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------</td>
<td>-----------</td>
<td>-----------</td>
</tr>
<tr>
<td>Bowling</td>
<td>58.594</td>
<td>56.1199</td>
<td>53.773</td>
</tr>
<tr>
<td>Boxing</td>
<td>58.168</td>
<td>51.2638</td>
<td>57.504</td>
</tr>
<tr>
<td>Breakout</td>
<td>7.897</td>
<td>4.73623</td>
<td>6</td>
</tr>
<tr>
<td>Carnival</td>
<td>2223.66</td>
<td>1487.83</td>
<td>2258.73</td>
</tr>
<tr>
<td>Centipede</td>
<td>5767.869</td>
<td>5491.39</td>
<td>5793.79</td>
</tr>
<tr>
<td>Chopper Command</td>
<td>964.7</td>
<td>971.983</td>
<td>962.5</td>
</tr>
<tr>
<td>Crazy Climber</td>
<td>34649.7</td>
<td>34285.2</td>
<td>11300</td>
</tr>
<tr>
<td>Defender</td>
<td>4857.45</td>
<td>4370.79</td>
<td>3797.9</td>
</tr>
<tr>
<td>Demon Attack</td>
<td>602.66</td>
<td>618.874</td>
<td>800.155</td>
</tr>
<tr>
<td>Double Dunk</td>
<td>-23.866</td>
<td>-23.7195</td>
<td>-23.102</td>
</tr>
<tr>
<td>Elevator Action</td>
<td>516.6</td>
<td>84.0567</td>
<td>180.4</td>
</tr>
<tr>
<td>Enduro</td>
<td>131.905</td>
<td>113.173</td>
<td>134.594</td>
</tr>
<tr>
<td>Fishing Derby</td>
<td>-99</td>
<td>-98.9644</td>
<td>-99</td>
</tr>
<tr>
<td>Freeway</td>
<td>19.551</td>
<td>20.3987</td>
<td>19.847</td>
</tr>
<tr>
<td>Frostbite</td>
<td>147.06</td>
<td>128.432</td>
<td>134.46</td>
</tr>
<tr>
<td>Gopher</td>
<td>505.22</td>
<td>463.687</td>
<td>541.16</td>
</tr>
<tr>
<td>Gravitar</td>
<td>582.85</td>
<td>482.003</td>
<td>500</td>
</tr>
<tr>
<td>H.E.R.O</td>
<td>11215.26</td>
<td>8980.43</td>
<td>12025.6</td>
</tr>
<tr>
<td>Ice Hockey</td>
<td>-8.349</td>
<td>-8.62203</td>
<td>-7.599</td>
</tr>
<tr>
<td>James Bond</td>
<td>323.75</td>
<td>234.318</td>
<td>431.1</td>
</tr>
<tr>
<td>Journey Escape</td>
<td>-29193.2</td>
<td>-29050.5</td>
<td>-29077.4</td>
</tr>
<tr>
<td>Kangaroo</td>
<td>573.8</td>
<td>558.987</td>
<td>600</td>
</tr>
<tr>
<td>Krull</td>
<td>4023.99</td>
<td>3120.23</td>
<td>4427.38</td>
</tr>
<tr>
<td>Kung Fu Master</td>
<td>7266.5</td>
<td>4259.88</td>
<td>7842.6</td>
</tr>
<tr>
<td>Montezuma Revenge</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Ms. Pac-Man</td>
<td>1353.73</td>
<td>986.353</td>
<td>1372.82</td>
</tr>
<tr>
<td>Name This Game</td>
<td>1929.22</td>
<td>2176.5</td>
<td>2011.97</td>
</tr>
<tr>
<td>Phoenix</td>
<td>1735.96</td>
<td>1294.02</td>
<td>943.31</td>
</tr>
<tr>
<td>Pitfall</td>
<td>-6.076</td>
<td>-7.23423</td>
<td>-24</td>
</tr>
<tr>
<td>Game</td>
<td>Score Last 1000</td>
<td>Score All 30000</td>
<td>Score Last Parameter Vector</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------</td>
<td>-----------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>Pong</td>
<td>-21</td>
<td>-20.9973</td>
<td>-21</td>
</tr>
<tr>
<td>Pooyan</td>
<td>1289.34</td>
<td>1139.66</td>
<td>653.845</td>
</tr>
<tr>
<td>Private Eye</td>
<td>98.425</td>
<td>92.0768</td>
<td>99.9</td>
</tr>
<tr>
<td>Q*Bert</td>
<td>514.3</td>
<td>637.579</td>
<td>450</td>
</tr>
<tr>
<td>River Raid</td>
<td>1655.3</td>
<td>1580.29</td>
<td>1601.67</td>
</tr>
<tr>
<td>Road Runner</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Robotank</td>
<td>30.426</td>
<td>31.2355</td>
<td>30.896</td>
</tr>
<tr>
<td>Seaquest</td>
<td>628.98</td>
<td>521.613</td>
<td>606.76</td>
</tr>
<tr>
<td>Skiing</td>
<td>-15037.803</td>
<td>-15496.4</td>
<td>-13726.5</td>
</tr>
<tr>
<td>Solaris</td>
<td>2980.96</td>
<td>2067.79</td>
<td>2166.08</td>
</tr>
<tr>
<td>Space Invaders</td>
<td>378.91</td>
<td>336.995</td>
<td>446.18</td>
</tr>
<tr>
<td>Star Gunner</td>
<td>934.1</td>
<td>930.707</td>
<td>1000</td>
</tr>
<tr>
<td>Tennis</td>
<td>-23.369</td>
<td>-23.3577</td>
<td>-24</td>
</tr>
<tr>
<td>Time Pilot</td>
<td>3592.1</td>
<td>3249.94</td>
<td>3661.7</td>
</tr>
<tr>
<td>Tutankham</td>
<td>108.568</td>
<td>108.564</td>
<td>109.136</td>
</tr>
<tr>
<td>Up and Down</td>
<td>4017.1</td>
<td>4159.26</td>
<td>4992.69</td>
</tr>
<tr>
<td>Venture</td>
<td>928.5</td>
<td>716.343</td>
<td>1400</td>
</tr>
<tr>
<td>Video Pinball</td>
<td>13514.086</td>
<td>12562.9</td>
<td>12297.6</td>
</tr>
<tr>
<td>Wizard of Wor</td>
<td>505</td>
<td>386.05</td>
<td>542.2</td>
</tr>
<tr>
<td>Yars Revenge</td>
<td>8482.979</td>
<td>7429.03</td>
<td>7245.96</td>
</tr>
<tr>
<td>Zaxxon</td>
<td>2244.3</td>
<td>1855.14</td>
<td>2380.7</td>
</tr>
</tbody>
</table>

Table A.1: Results of Sarsa training for the last 1000 episodes and for all episodes (30000), along with the score obtained with the last computed parameter vector.
Bibliography


[20] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis,


[23] Levente Kocsis, Csaba Szepesvári, and Jan Willemsen. Improved Monte-Carlo Search.


