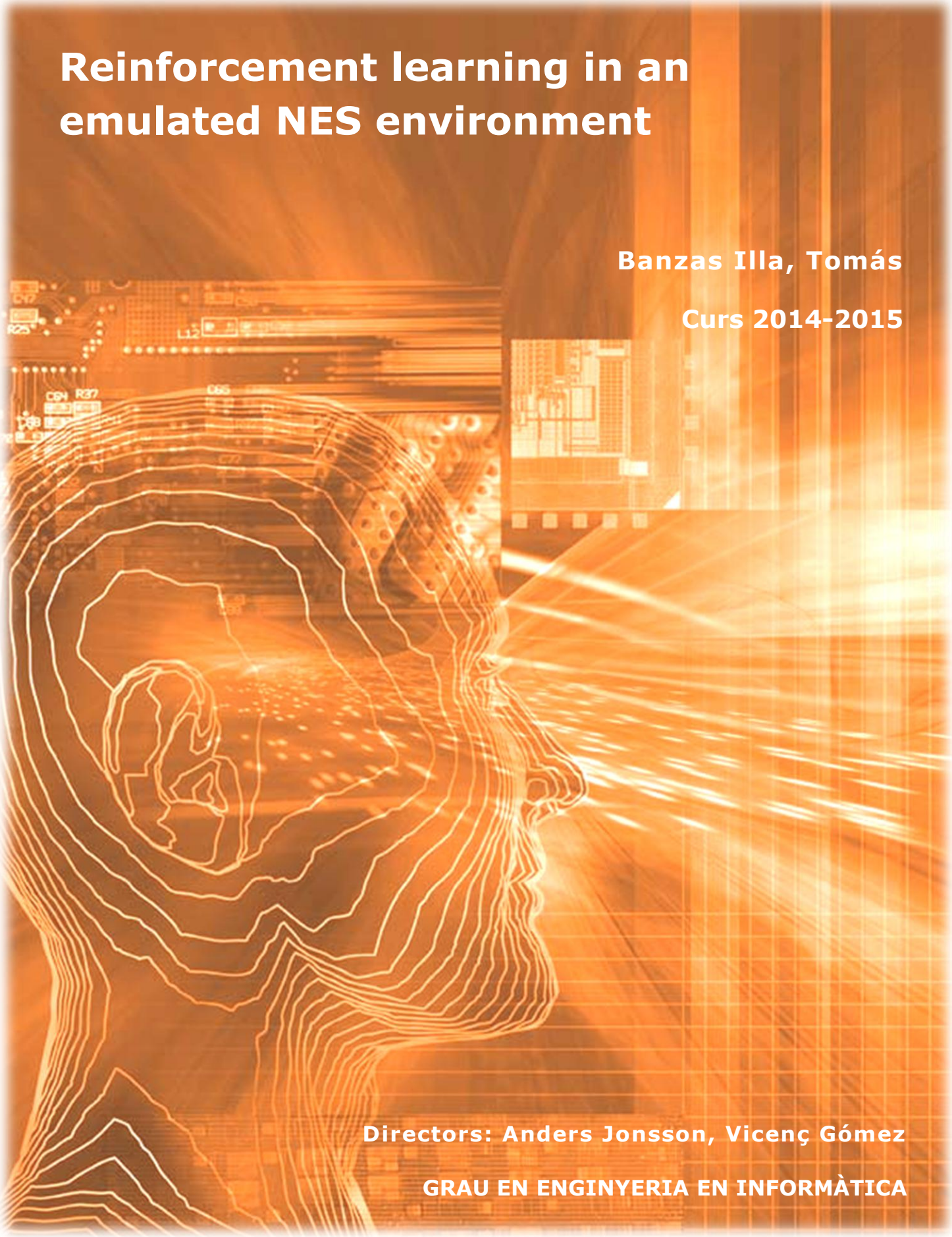
Reinforcement learning in an emulated NES environment

Banzas Illa, Tomás

Curs 2014-2015



Directors: Anders Jonsson, Vicenç Gómez

GRAU EN ENGINYERIA EN INFORMÀTICA

upf. Universitat Pompeu Fabra Barcelona
Escola Superior Politècnica

Treball de Fi de Grau

Reinforcement learning in an emulated NES environment

Tomás Banzas Illa

TREBALL FI DE GRAU

Bachelor's degree in Computer Sciences

ESCOLA SUPERIOR POLITÈCNICA UPF

2015

DIRECTOR DEL TREBALL

Anders Jonsson
Vicenç Gómez

To my family, friends and to those who sleep during the day.

Acknowledgement

I would like to extend a thankful acknowledgment to my friend Facundo Nahuel Martin who helped with some debugging and donated some processing hours when my computer was dead.
Also to my friend Esteban Rodríguez who provided help into understanding and laying down some mathematical notions that eluded me.

Abstract
A short review and comparison of Q-Learning, Function Approximation by gradient descent and Monte Carlo Tree Search algorithms, implemented to run on an environment based on an emulation of the Nintendo Entertainment System video gaming console. The Nintendo Entertainment System and its catalogue of games provide a multitude of scenarios to research learning algorithms. Different states and rewards are produced in real-time using memory snapshots provided by an emulator running different games. Although the state space of Nintendo Entertainment System is much larger than that of an Atari, Monte Carlo Tree Search is still able to learn useful policies.

Resumen
Un breve análisis y comparación de los algoritmos de Q-learning, Function Approximation por descenso de gradiente y Monte Carlo Tree Search, implementados para correr en un entorno basado en una emulación de la consola de videojuegos Nintendo Entertainment System. La Nintendo Entertainment System y su catálogo de juegos proveen de una multitud de escenarios para investigar algoritmos de aprendizaje. Diferentes estados y recompensas son producidos en tiempo real usando capturas de memoria proveídas por un emulador ejecutando distintos juegos. Aunque el espacio de estados de la Nintendo Entertainment System es mucho más grande que el de una Atari, Monte Carlo Tree Search es aun capaz de obtener algunos resultados.

Preface

The Nintendo Entertainment System (NES) has been always dear to me. Working on this small project allowed me to learn and understand a little of the inner workings of the games of this interesting machine. And also to waste an extensive amount of time playing games that are several decades old and can only be enjoyed through nostalgia googles (luckily, mines are extra-thick.)

When picking a subject for my final degree project I already wanted to do something related to artificial intelligence and having seen some popular attempts at solving NES games I decided to give it a shot of my own.

On the following pages we will discuss the interest of making the whole NES game catalogue available for reinforcement learning research, how NES memory usually works, we talk about how we created a NES based environment and how to create reward and terminal functions for any game to test some player agents on them.

Index

Figure list

1. INTRODUCTION

On this project, we built an environment to allow reinforcement learning agents to be tested against games from the NES.

1.1 Motivation

When asked "Why?" we could probably reply "Because watching the computer struggling to learn to play on its own would be quite amusing" but that probably wouldn't be enough to explaining the advantages of having such an environment.

Each videogame represents a different challenge. Some games might have hundreds of different actions at each step, and expect the player to react really fast to short term obstacles. Others might give a few, but require the player to think ahead.

The Nintendo Entertainment System official catalogue holds hundreds of titles including but not limited to puzzle, platformers, racing games, space shooter and sport games.

In Galaxy patrol, one of the titles we test with, the player has evade dangerous killer obstacles but also to do so looking ahead as to not put himself in a place where he can't replenish energy.

In Super Mario Bros, a platformer, the player has 300 seconds to reach the other end of the level or face a negatively rewarded termination. Meaning that you can easily have "already lost" the game and still have a lot of time left if you mismanaged it

Constructing a generic artificial player that is able to learn and generalize good performance on different videogame genres is a challenging task not yet solved. If we can represent multiple games using the same standard, we would be able to see how fast learning algorithms learn in different scenarios without having to waste an extensive effort in adapting them for each particular case (or wasting time developing these scenarios if we can actually transform games into scenarios to test). This could potentially give us another tool to gauge the ability of a certain agent in a determinate situation against different kinds of challenges. On chapter 5 we look in detail at some of the games we used on this project and the results that different agents managed on them.

While we only worked with five games, the methods explained on this document should be easy to apply to most of the Nintendo Entertainment System catalogue.

1.2 The problem

The problem that we took upon ourselves has been finding an adequate representation of the agent and the way it communicates with the environment. Not only how the agent controls the environment, but also which kind of observations can be given to the agent

and how to determine if the actions of the agent deserved a reward or not.

For this project, the observations the agents receive will be snapshots of the onboard RAM memory of the games (or rather their emulated counterpart) instead of what would a human player would receive, images and sound.

In chapter 4 we discuss the memory management system of the NES works and how we can find meaningful addresses in these memory snapshots to calculate which states deserve rewards and which states are terminal.

2. BACKGROUND

2.1 Introduction

This chapter is mostly dedicated to review the scenario in which this project comes in, the documentation we have available and the tools we have at our disposal to solve the problems.

The latter sections are mostly centered on the impact of the Nintendo Entertainment System in the world, emulation software, the thin line of legality of getting ROMs and some sources recommended to understand more about the system.

2.2 Reinforcement Learning

Most of the reinforcement learning concepts of this project have been based on the algorithms presented on *Reinforcement Learning: An Introduction by Richard S. Sutton and Andrew G. Barto*[1]. In particular, the Q-learning and Gradient-Descent function approximation agent mentioned on the next chapters.

Reinforcement Learning is an area of the field of machine learning that looks into how to generate agents able to choose actions in an environment in such a way that the accumulated rewards are maximized. In reinforcement learning our algorithms don't have an accurate idea of what is a good or bad move, and the only lead is being able to relate observations of the environment (observations that might not be complete or accurate) with some provided rewards. The agent then uses the rewards to extrapolate the value of each new observation as to be able to determine which action would be more convenient (not always trying to get the best rewards, but also doing exploratory moves with the possibility of finding even better rewards).

2.3 Monte Carlo Tree Search (MCTS) research hub

"Monte Carlo Tree Search (MCTS) is a method for making optimal decisions in artificial intelligence (AI) problems, typically move planning in combinatorial games. It combines the generality of random simulation with the precision of tree search.

Research interest in MCTS has risen sharply due to its spectacular success with computer Go and potential application to a number of other difficult problems. Its application extends beyond games, and MCTS can theoretically be applied to any domain that can be described in terms of {state, action} pairs and simulation used to forecast outcomes." [2]

Our third agent, a Monte Carlo Tree Search agent, is based on the code explained in the MCTS research hub. MCTS builds a tree where each node represents a state, in our case being the product of performing a certain chain of actions.

2.4 Nintendo Entertainment System

Figure 1 - By Evan-Amos (Own work) [Public domain], via Wikimedia Commons

The Nintendo Entertainment System (referred as NES in the rest of this work) is a gaming console released by Nintendo in 1983. The machine had worldwide impact and even managed to survive the videogame crash of the 80's where many other systems failed.[3]

There's a big community of enthusiasts behind the NES, which includes several forums, wikis and IRC[1] channels dedicated to the subject, even decades after its conception. A sizeable part of these communities dedicated to emulation, which consists of software able to play the old titles of the machine (or even new unofficial homebrew titles).

There have been also some efforts to hold AI competitions[4] on Nintendo games which shows there's some interest on the subject. On this project we will explore one of the possibilities that emulation software offers for reinforcement learning.

2.5 Emulation software

Essentially, emulation software allows a computer system to (as it name implies) emulate another system. In the particular case of video game console emulators, the emulated systems are gaming consoles. Usually having a particular hobbyist and advanced public as a target means that console emulators usually come bundled with lots of features and tools that were not available on the original system, either for regular playing, tool assisted speed runs(TAS) or hacking/programming purposes.

[1] IRC: Internet Relay Chat. A popular protocol of chat communication.

Most complete emulators allow to inspect and alter the memory addresses of the emulated system, tinker with controls and display(such as automatically repeating buttons or button-combinations, or visual filters), slow down, pause or speed up the game speed and save and load the game status at any moment, all of this in the middle of emulation.

Figure 2- Galaga by Namco, running on the FCEUX NES emulator

For this project we focus to center ourselves on the FCEUX [5] emulator which has all the mentioned capabilities. FCEUX is open source and available for Windows and Linux.

One of the interesting features of FCEUX is that it comes packed with the LUA scripting language, which facilitates interaction by abstraction of some high-level features.

It seems attractive to develop an environment which would allow reinforcement learning agents to be tested against NES games on an emulator, because the same interface would allow to test all the NES catalogue with very little adaptations on a platform that is active and supported.

There are games with time limits, puzzles, games with limited moves and infinite ones. These multiple games mean multiple scenarios with different restrictions and possibilities.

2.6 ROMS

a) ROMs and emulation

Unlike most up-to-date gaming consoles that use discs, NES games came in cartridges that held the read-only memory. These pieces of hardware sometimes even were able to expand the basic functionality of the console (extra memory for graphics in the case of the SNES[2], some cartridges even came with extra processors to perform 3d calculations!).

Console emulators need a file containing the dumps of the read-only memory of the original hardware. In the case of NES, most emulators work with the **nes** ROM image format.

b) Legal Issues with ROMs

Emulation in general stands in a pretty gray area in respect of legality. While generally there's certain leeway allowed for preservation and educational purposes, the distribution of ROM files is illegal in many countries, just as playing a ROM without having the original cartridge.

However, generating ROMs for personal use is arguably legal. The cartridge dumping process for the NES, that is, the process in which one generates a ROM image, will not be explained in this document.

c) Homebrew ROMs

There also ROMs programmed by enthusiasts, often hacks of original games, sometimes completely new ones. These titles actually don't belong to the official system catalogue, though are fully compatible with emulators.

There are several websites, such as http://www.romhacking.net/[6] that provide users with dozens of these games and tools to generate them.

2.7 NES Documentation & support

While not needed to completely understand the work in this project, the Nesdev wiki[7] makes available all sorts of guides and information to have a more complete knowledge of the NES, there's also an active community on its forums. Enthusiasts gather there to share their homebrew games and discuss questions.

2.8 NES memory addresses

To write terminal and reward functions, we need to find useful memory addresses in the NES RAM. While it's perfectly possible to isolate these memory positions just using the tools provided by FCEUX (a process we explain briefly in *4.1.b How the memory addresses are found*) many popular games already have some of their memory positions publicly identified.

Addresses that refer to values such as scores, levels, quantity of lives left are usually used to make cheats or hacks, but can help us to define our reward/terminal functions.

[2]SNES: Super Nintendo Entertainment System, the gaming console by Nintendo that succeded the NES.

NES hacker wiki[8] supplies with many of these memory addresses.

Another good resource is the FCEUX documentation[9], which not only explains how to use the software, but a lot of technical information about how the NES hardware works.

3. METHODOLOGY

On this chapter we discuss what has been worked on this project,
We discuss the RL-NES environment, the reward and terminal states and the agents.
In the last section we mention related work.

3.1 RL-NES environment

We implemented a reinforcement learning environment in C++, capable of running an instance of the FCEUX emulator and communicates through LuaSocket[3]. The environment is able to open a game and sending actions to it as well ast also to tell the emulator to manage save states[4].

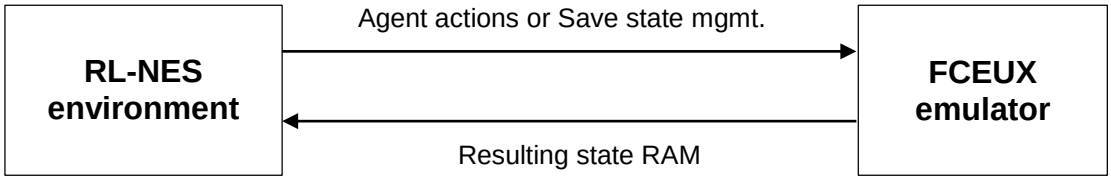


Figure 3 - RL-NES cycle repeats each frame of gameplay

At every frame of gameplay the RL-NES receives an action from an agent and sends it to the emulator, then the emulator in turn executes the action and sends the emulation RAM to the environment. The environment itself doesn't preserve state information (though it manages the emulator saves states as per agent requests) giving the agents full control as to how observations should be stored. For example, on our Q-learning Agent every state is saved as a key of a table to get the state-action values, while in our function approximation one, we save none.

We must take into consideration that while the refresh rate of the NES is 60 frames per second, which means 1 second of gameplay at the intended speed would produce 60 memory snapshots per second. However, given the emulator capabilities and modern processing power the emulator can be forced to overclock and run the games dozens of times faster, producing several hundreds of states per second. On the computer we tested the agents, the emulator could reach easily between 800 and 1200 frames per second. While this is good, because our agents need not to be limited to learn the games at normal speed, if we were to store fully all the states we encounter, we would need a lot of memory.

[3] LuaSocket is a Lua library that allows to communicate through TCP/IP. Our environment stablishes a rudimentary protocol with a Lua script that allows it to send actions and receive the state.
[4] Save state: Emulators are able to save and load the game status at any particular frame, even where the original game would have not allowed so, by storing the RAM of the game.

There are several ways in which this could be handled. For example, Most NES games don't use all memory positions. While not implemented for this project, it wouldn't be hard to observe which memory positions each game really uses and ditch those not needed. Dr. Tom Murphy[12] describes a method in which a tabula rasa state is created from the most common values of each memory position, and then states are described as the addresses that don't match that basis state.

3.2 Rewards and Terminal states

The RL-NES environment decides whether every state that is processed is a terminal state or a state that produces a reward these two functions are game-dependent and we have implemented them for four official NES titles: Super Mario Bros, Galaga, Pacman) and one homebrew one (Galaxy Patrol). However, expanding the list is quite easy.

3.3 Agents

We implemented three agents to test the environment. The implementation of these three is discussed in the next chapter.

a) Q-learning agent

A Temporal Difference control agent that is able to learn the optimal policy in spite of the agent actions (off-policy), as long as enough exploration is granted. It has been rightfully lauded as *one of the most important breakthroughs in reinforcement learning*[10].

b) Gradient-Descent

This agent performs a generalization function based on the gradient descent method. In essence, this agent should be able to produce some useful values for states it never encountered before based on observations from similar states. Function approximation allows to occupy less memory and require less time to be filled, since it doesn't actually stores all the states it encounters but rather just haves a table of fixed size: *actions x features*.

c) Monte Carlo Tree Search

The third agent produced better results than the other two, implementing Monte Carlo Tree Search. This agent builds a tree, adding one node at a time per simulation. It was for this one that we added the functionality of loading and saving states to the environment. MCTS (Monte Carlo Tree Search) simulates states arbitrarily so being able to jump between states was needed the agent to actually work.

3.4 RL-Glue interface

The environment we implemented follows the RL-Glue[11] interface, which provides an standard of functions to implement agents, environments and experiments even if they're written in different languages.

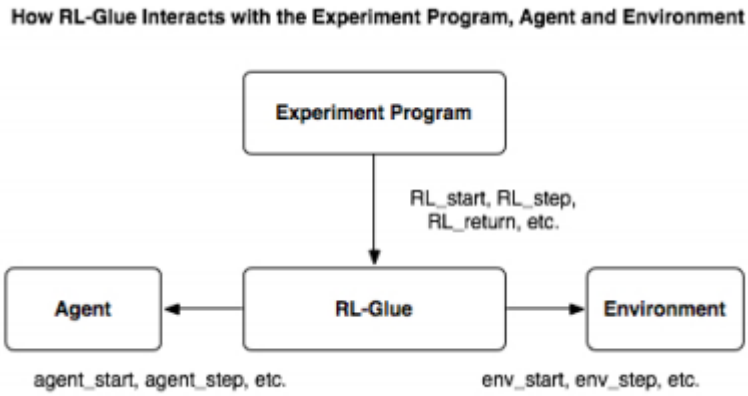


Figure 4 - The RL-Glue Standard. Arrows indicate function call direction

3.5 Related work

The idea of this project, running learning algorithms on emulators, is certainly not a novelty:

a) Playfun/Learnfun

Dr. Tom Murphy presents a simple, generic method of automating the play of NES games[12]. The results are quite impressive as his automated playthroughs managing to come up with movements that probably wouldn't be possible for a regular human.

b) Arcade Learning Environment.

"The Arcade Learning Environment (ALE) is a simple object-oriented framework that allows researchers and hobbyists to develop AI agents for Atari 2600 games. It is built on top of the Atari 2600 emulator Stella and separates the details of emulation from agent design".[13]

ALE provides for a similar functionality to ours (though much more polished), however, it does it with an Atari 2600 emulator, which had only 128 bytes of RAM in comparison to the 2048 bytes of the NES.

4. IMPLEMENTATION

On the first section of this chapter, we discuss how the RL-NES environment is put together, how NES memory is organized and how based on this information, we can find interesting memory addresses to write terminal and reward functions.

The second section explaining the three agents we have programmed to test against the environment.

4.1 RL-NES environment in detail

Following the RL-glue standard we actually have, for the most part, a good lead of how an environment is expected to behave. RL-glue help us to streamline and evade certain bad ideas, on the other hand, the separation between environment and agent and the protocol it enforces makes a little bit harder to pass data between agent and environment.

The most important part of the environment is the step functions that the RL-glue interface suggests: *env_start* and *env_step*.

At the beginning of each episode, *rl_glue* executes *env_start*, which initializes the terminal and reward functions for the chosen game and tells the emulator to restart. All done, *env_start* receives the first state memory and returns it.

Then, during the rest of the episode the flow is passed between the environment and the agent: the agent responds to state memory it receives with an action, and the environment passes the action to the emulator in *env_step* along any reward finds with the reward function. The episode stops when the environment detects with the terminal function that the game is over, passing one final return to the agent with just the reward.

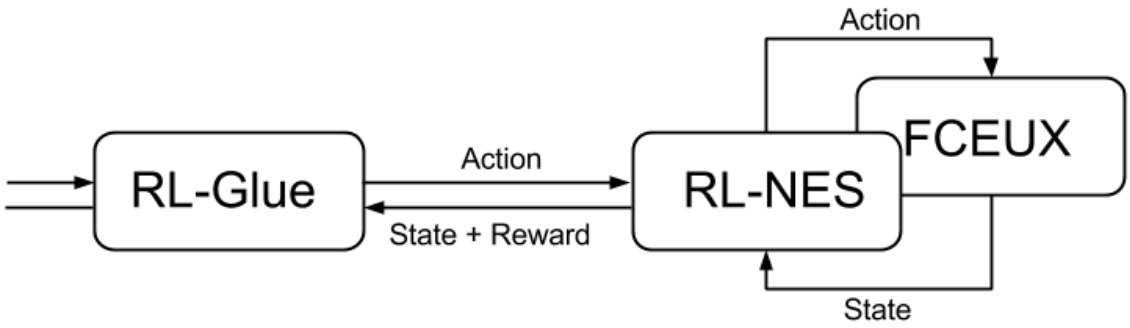


Figure 5 - RL-Glue / RL-NES interaction

a) Memory

To stop an episode (or a simulation, in the case of the MTCS algorithm) we would need

to know whether a state is representative of a game over scenario or not.

Let's entertain the following hypothetical situation:

If we were programming ourselves a game that an agent has to solve, if we were developing a game were the player should be rewarded for going up, it would be easy to do the rewards as a differential of a certain height variable. That is, if we wanted to reward the agent for managing for going up. Perhaps if we wanted to stop the episode, when the agent reaches its destination (or its game over) we could just test if the aforementioned height variable achieved a maximum value or maybe we want the game to stop when the player has 0 lives because he fell into a pit too many times.

In this case, it happens exactly the same, these ideas would probably be quite similar for any game-like scenario at least. However, as we receive the memory from the emulator, we really don't have such variables labeled, it's just a cold and ruthless vector of 2048 bytes… or is it?

While, the task of finding which values are interesting it's not as straightforward as we could wish it to be, the memory snapshot is not devoid of semantics readily to be exploitable and the enthusiast community have already figured most of the tricks for us.

b) How the memory addresses are found

The 2048 bytes of NES onboard RAM are organized in 8 "pages", blocks of 256 bytes each[14].

Block 0	$00xx
Block 1	$01xx
Block 2	$02xx
Block 3	$03xx
Block 4	$04xx
Block 5	$05xx
Block 6	$06xx
Block 7	$07xx

In the tools available in FCEUX, the addresses of these blocks are referred in hexadecimal, each block having addresses from 00 to FF (and given the addresses hold bytes, each one of these values have 256 possible values from 00 to FF again).

To anyone that used similar tools (either for cheating or for more noble purposes like research) the following part will not be much of a mystery. The first tool is the Hex Editor. This one allows us to watch the values of the memory change in real time, and even alter the values by hand!

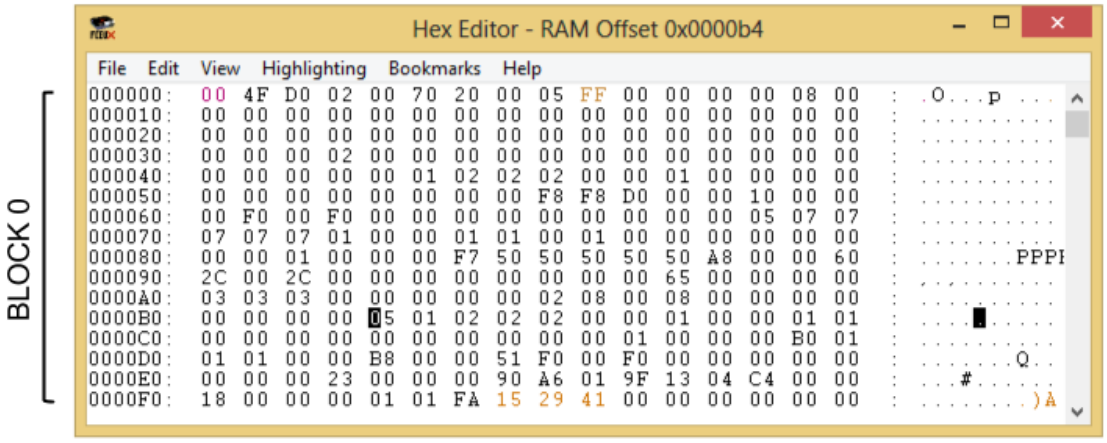


Figure 6 - FCEUX Hex Editor, Block 0 of Super Mario Brothers

We can guess many of the meanings of the positions just by watching and playing ourselves, yet there are further tips that can help us find more values faster. Addresses on the same row will usually hold the same kind of data. For example when moving left or right in Super Mario Brothers we will easily see how the address $0086 holds a component of horizontal position of the main character. When enemies enter on the screen, we will see that their horizontal positions will be on the same row.

Also, almost always the values like scores or lives will be on the first pages, so it's not hard to spot them just watching the values change.

The other interesting tool is the RAM search. This tool allows us to search directly in all the addresses by value. Once matching addresses are found it will inform us of any new values. So, even if we were looking for the address that holds the remaining lives and several addresses match the value of the displayed lives at some point, we will know when the value decreases which one is the correct.

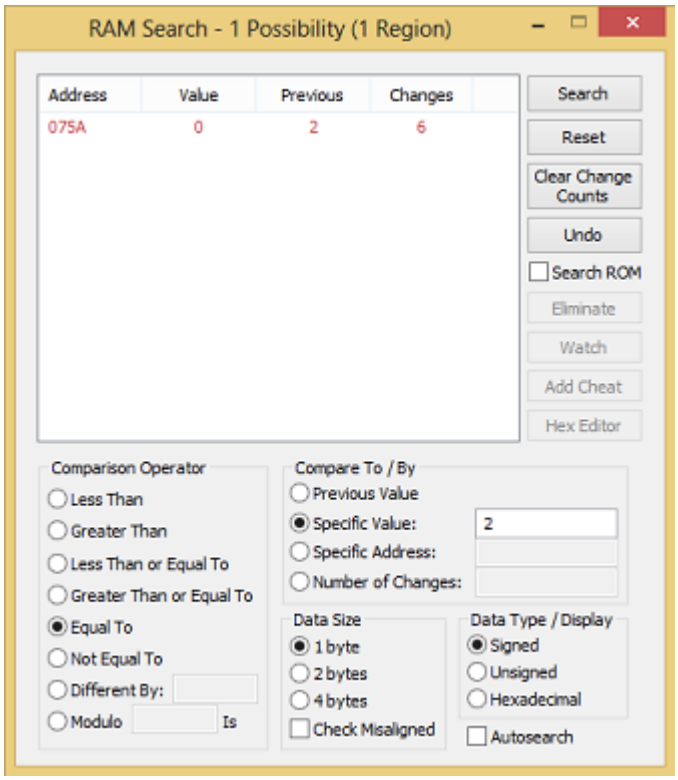


Figure 7 - FCEUX RAM search

c) Reward and Terminal functions

Having the relevant addresses written down, there's not much more complexity into how to write the reward functions. Essentially we need a function that receives as a parameter the state, look up the positions (we will need to change the hexadecimal values and positions into integers) and check up their values.

As an example, to calculate the reward of Galaga, we look up at the addresses 224-230, which hold each one a character from the score. We only need to look up at the values the addresses had on the last frame and see if they went up to decide whether the agent deserves a reward or not.

4.2 Agents

a) Q-Learning Agent

Basically the same as the algorithm described chapter 6.5 of *Reinforcement Learning: An Introduction*[10], this first agent performs off-policy, one-step Q-learning.
Each step, the agent receives the current state $\boldsymbol{s}$ of the game and chooses an action $\boldsymbol{a}$ with an ε-greedy policy from the action-value function, $\boldsymbol{Q}$**.** In our case, the values of the function $\boldsymbol{Q}$ are stored in an unordered map that has the oncoming states as key, and the actions as columns.

After sending our action to the environment, we receive the new state $\boldsymbol{s'}$ (result of the environment applying the chosen action) and a reward $\boldsymbol{r}$. Using the chosen action, and state and reward, we update the values of the $\boldsymbol{Q}$ function using the following formula:

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

This essentially means that our current state values will be affected with future state values eventually, playing with the step-size parameter $\boldsymbol{\alpha}$ and discount-rate parameter $\boldsymbol{\gamma}$ will determine how aggressively the values are updated.

$Q(s, a)$ is initialized to 0
Repeat(for each episode):
    Initialize s to initial state
    Repeat (for each step of episode)
        Choose a from possible actions using ε-greedy with $Q(s, a)$
        Take action a, receive r and s'
        Update $Q(s, a)$
        $s \longleftarrow s'$
    until s is terminal

Figure 8 - Q-learning pseudocode

b) Gradient-Descent

Loosely based on the linear, Gradient-Descent[15] found on the same book. In contraposition to the Q-learning agent, this agent doesn't store all the states it meets, instead it has a table of fixed size.

Each step the agent receives the 2048 bytes of the state and calculates the value of each possible action given said state. The value function $\boldsymbol{Q}$ is now calculated like this:

$$Q(s, a) = \theta_1^a \phi_1^s + \cdots + \theta_k^a \phi_k^s$$

Essentially we have a vector $\boldsymbol{\theta_a}$ that holds a value for each possible feature (we will now explain this) for each possible action $\boldsymbol{a}$. To check the value of a state-action pair we should extract a vector of features $\boldsymbol{\phi^s}$ from the state. We tried in two different ways:

The first one was generating a feature for each possible value of each byte$[0 - 255]$. This results in a vector $\boldsymbol{\phi^s}$ that hold in each position a Boolean that indicates whether the corresponding position on $\boldsymbol{\theta_a}$ should be summed or not. For this to work, both $\boldsymbol{\theta_a}$ and $\boldsymbol{\phi^s}$ should store 524288 values (that is, 2048*256). However, when actually programming this, instead of doing a vector of Booleans for $\boldsymbol{\phi^s}$, we can just store the active positions and iterate through them to make the ordeal cheaper in the long run (and on games with lots of actions not doing this can really slow down everything!)

The second one (the one we ended up going with) consisted of using as feature each individual bit from each byte. For example if the byte 0 has the value 16, it would correspond to [0, 0, 0, 1, 0, 0, 0, 0]. Now, for each byte multiple positions of $\boldsymbol{\theta_a}$ would become part of the summation. This time means that our vectors $\boldsymbol{\theta_a}$ and $\boldsymbol{\phi^s}$ should store 16384(yes, 2048*8) values.

Having the value function, given a state we can compare the value of each possible state-action pair. Using an ε-greedy policy we either select the best action or we explore, and the environment will in turn provide us with the reward $\boldsymbol{r}$ and state $\boldsymbol{s'}$. Given this we update $\boldsymbol{\theta_a}$ as follows:

$$\theta_a \leftarrow \theta_a + \alpha[r - Q(s,a) + \gamma Q(s',a')]\nabla_{\theta_a} Q(s,a)$$

That is:

$$\theta_a \leftarrow \theta_a + \frac{\overrightarrow{\phi^s} \times \alpha[r - Q(s,a) + \gamma Q(s',a')]}{\sum \overrightarrow{\phi^s}}$$

This means that we have to choose the next action before applying the values to θ_a. Having all these elements, we can put the pseudo code easily as follows:

Initialize θ *to* 0
Repeat(for each episode):
    Initialize s to initial state
    Initialize a to a random action
    Repeat (for each step of episode)
        Take action a and receive r and s'
        Choose a' from possible actions using ε-greedy with $Q(s',a')$
        Update θ_a
        $s \leftarrow s'$
        $a \leftarrow a'$
    until s is terminal

Figure 9 - Gradient Descent Function Approximation

c) Monte Carlo Tree Search

The third agent follows the basic algorithm of the MCTS hub almost down to the letter. We will detail how it works in our particular case, what adaptations we had to make and what problems we have found.

MTCS establishes a loop of four steps that iteratively builds a search tree of increasingly more quality each time the cycle is completed: **Selection**, **Expansion**,

Simulation and **Backpropagation**.

The root of our tree is the initial state of the game, while each children represents the resulting state of choosing one of the possible actions for a determinate game. In a game like *Super Mario Brothers*, where more actions are possible we have lots of possible children, yet in *Galaxy patrol* each node might only have three children nodes.

On the first step of our four-step cycle, we select one of the nodes of the tree applying recursively on the root a **selection** function that takes the best children node until it reaches a leaf.

In our case, the select the value we use the Upper Confidence Bounds formula[16] modified for our case:

$$Rewards + C \times \sqrt{\frac{\ln N}{n}}$$

$\boldsymbol{C}$ is a tunable bias parameter, $\boldsymbol{N}$ and $\boldsymbol{n}$ stand for the times the parent and the children have been visited respectively. Rewards in this case it's not the particular reward of a state, but actually the sum of the accumulated reward of the state itself and all its children found to the moment, including those rewards found during simulations.

For the following step, **expansion**, we look if the state is terminal. If not, we expand the node (again, each of the expanded children would represent one of the possible actions).

Having the node expanded, we choose randomly one of its children, and tell the environment to set the emulator at that state (if the state had never been reached we load the parent execute the children action and save the resulting state). We perform a **simulation** on that loaded state, effectively choosing our actions randomly till a condition is met. For this part we tried with different options, performing multiple simulations of the same state, and varying the maximum quantity of frames per simulation. Shorter simulations effectively allowed the agent to make the tree grow faster(less computation time wasted on simulations). However this robs the whole process a lot of precision.

After we finish our simulation step, we proceed to do **backpropagation**, this is quite straightforward, we sum the rewards acquired during the simulation recursively upwards from the leaf until we reach the root.

Now, the main issue here is that for each node, we have to save the state on the emulator to be able to reload it. FCEUX has a limit of dozens of thousands, which varies from system to system. That still might not be enough if we take in account the massive numbers number of states a single playthrough might visit. What we decided to do was, each time the tree reached certain size (the amount of nodes reaches the maximum savestates we can allow ourselves) we take the current best action and free the memory of the states that will not be revisited nor simulated anymore.

Even so, an alternative way of handling this would be exploiting further the deterministic nature of the emulation and encode states as a series of movements, we

could then eliminate the saved data of less frequented states but keep the branch which would allow us to rebuild the exact same states if they end up being needed in the future.

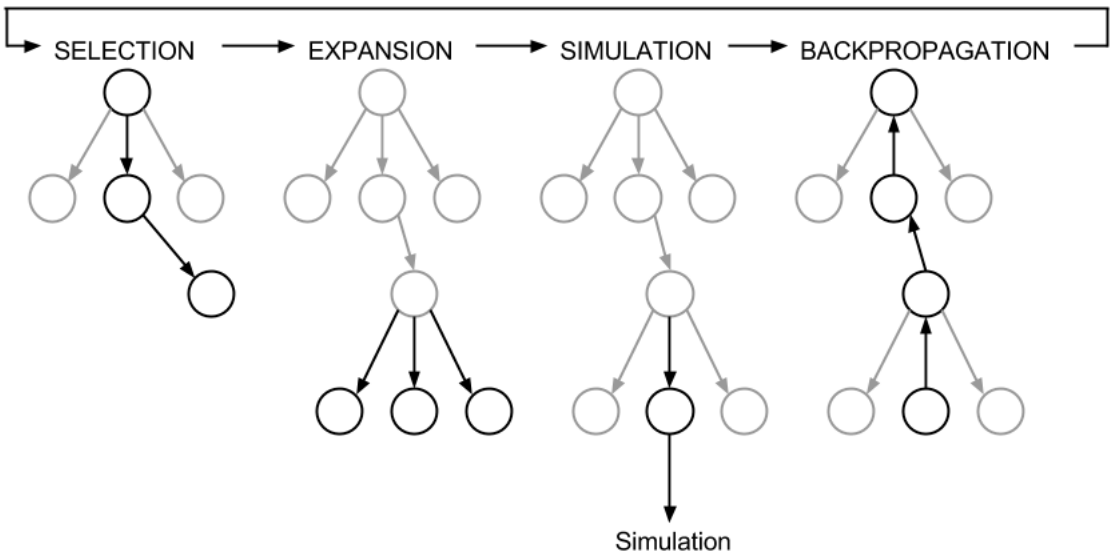


Figure 10 - MCTS

5. RESULTS

We will describe shortly each of the tested games, how the rewards and terminal states are calculated and describe how the agents progressed in each case.

For Q-learning and Function Approximation we generated graphs that compare rewards per episode, and the evolution of the median of reward as episodes advance.

The conventional NES controller has eight buttons (four directionals, start, select, A and B). For some NES games some buttons are not used or performed not so interesting behavior (pausing the game), we selectively tell the agents to ignore the actions related to said keys.

5.1 Galaxy Patrol

Galaxy Patrol by Michael Martin is a homebrew game of skill and reflexes. The player controls a ship with limited "fuel" units and should manage to survive for what seems to be an unlimited time.

On the scenery, from the bottom of the screen, both insta-kill space debris and fuel-replenishing pellets appear and the player should evade the former and collect the latter to extend his limited time.

The player can only move left and right, so we limited the actions accordingly.

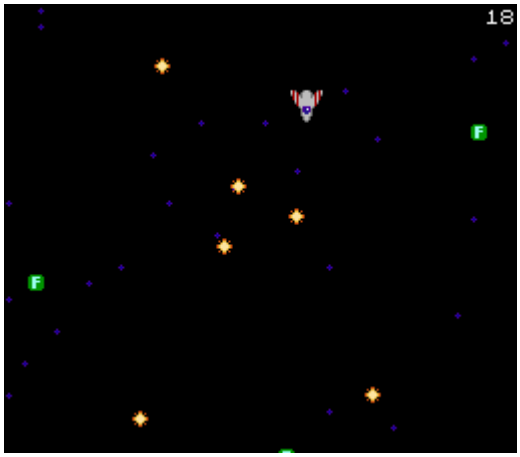


Figure 11 - Galaxy Patrol screenshot

a) Reward and terminal functions

The agents received +0.1 of reward each time they got fuel and were penalized with -1 when the episode ended with fuel depleted and -10 when they crashed against the debris.

A state is recognized as terminal when the player dies, either by collision or fuel depletion.

b) Results

Figure 12 corresponds with the Q-learning agent illustrating the rewards per episode. The two different ways into which an episode can end are plain to the naked eye: Either bad (fuel depletion, -1 point) or really bad (crashing against the space debris -10). Sadly, even in the simplest game there were so many states that with the nearly 8000 episodes practically all of the actions were still exploratory.

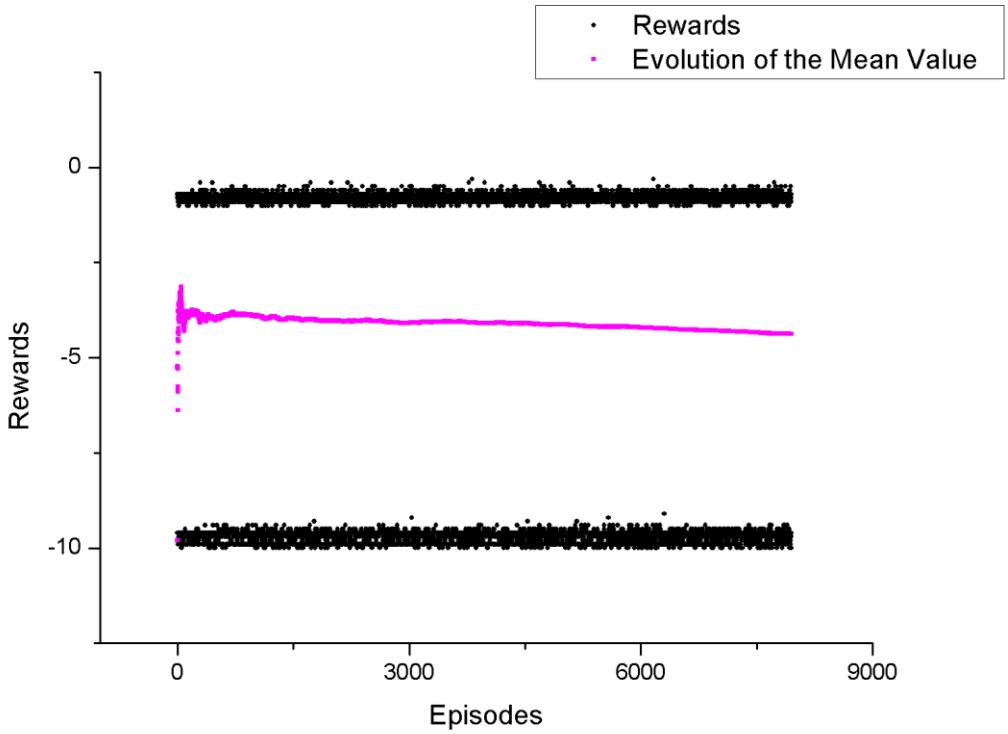


Figure 12 - Galaxy Patrol Results: Q-learning

For the function approximation agent things aren't that different, at least right now (Figure 13). Episodes can be clearly separated between those were the ship crashes and those were the combustible reaches 0. However we can observe that the Mean value goes down as long the episodes grow.

Figure 14 puts the two mean value evolutions toghether (since the coulds would be hardly readable for humans). It's quite dissapointing given that Q-learning which at this point is practically performing random actions performs so much better than the gradient descent function approximation.

Next is the is the MCTS agent. While difficult to compare with Q-Learning or function approximation, given that our MCTS agent doesn't behave in "episodes", has show more chances. Being allowed enough savestates and full simulations, the agent manages to work for a long time without getting stuck on dead ends. While we can shorten the simulations effectivelly making the agent build the tree faster, the quality is severely reduced and the fast completing tree ends up stuck in a dead end.

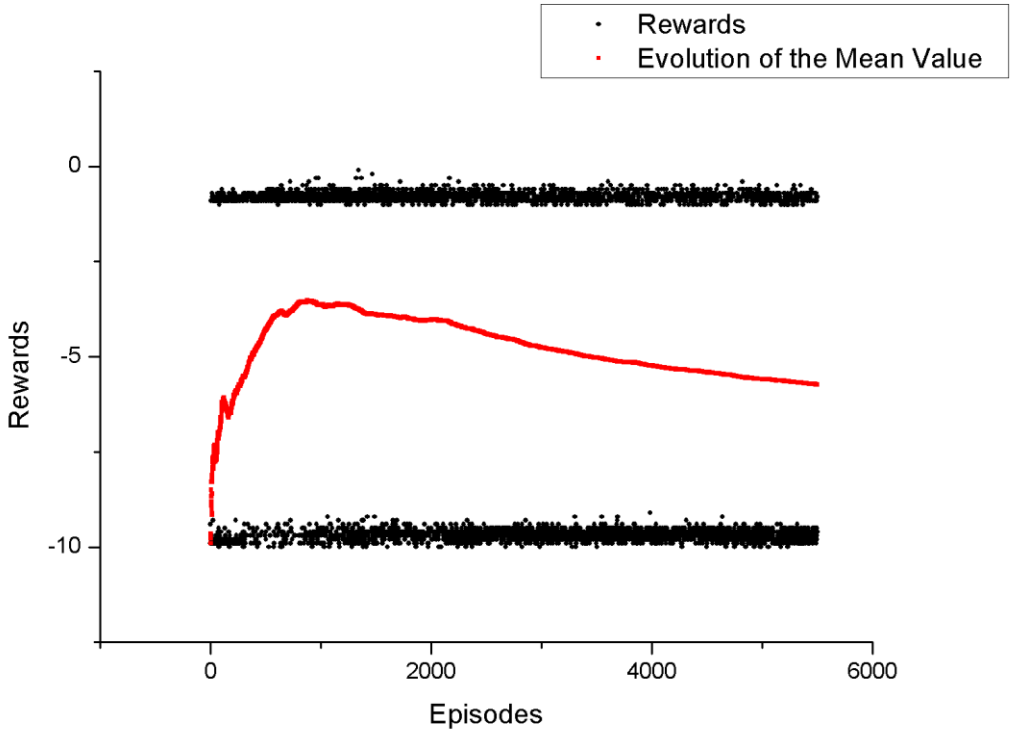


Figure 13 - Galaxy Patrol Results: Gradient Descent Function Approximation

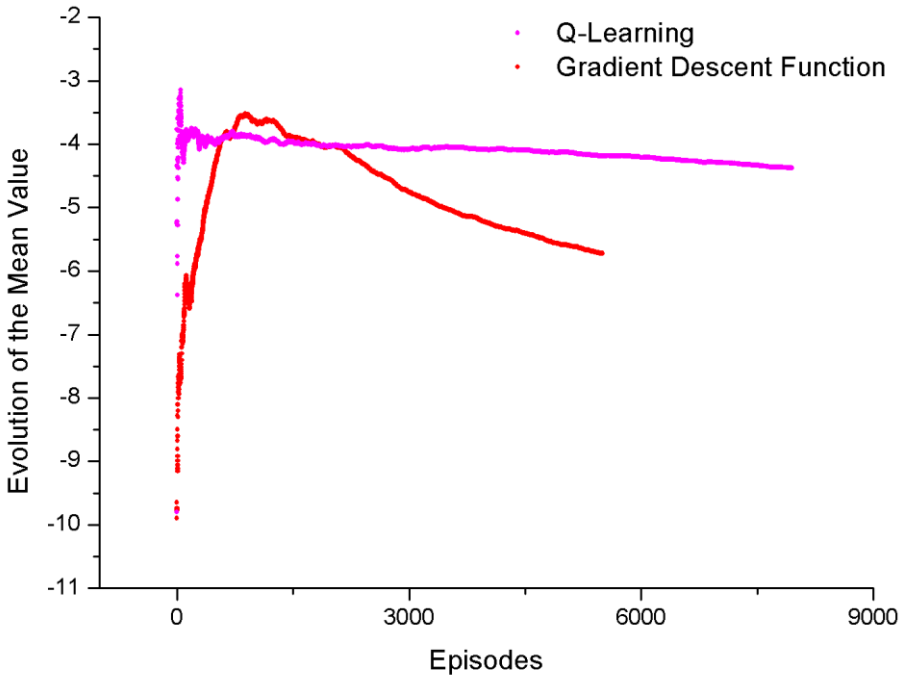


Figure 14 - Galaxy Patrol Results: Gradient Descent + Q-learning

5.2 Galaga

Galaga is a fixed space shooter, each level is filled with a squad of enemies which the player, positioned at the bottom off the screen, must shot down to complete and pass to the next round. The player has unlimited bullets, but only two can appear simultaneously on the screen. See figure 2.

The player can only move sideways and shoot, so we limited the actions accordingly, also we removed the pause button.

a) Reward and terminal functions

The agents received +1 of reward an enemy was hit. And since it's pretty easy for an agent to be stuck limitlessly on the same place, there's a really small penalization for being idle.

The episode ends when the player has no lives left.

a) Results

The small penalization of reward to incentivize the agent to do something other than standing idle in a corner made every single state have a reward of its own. For the Q-learning agent, which didn't need to save states with 0 value, that meant all states had to be saved. But our implementation wasn't able to store all of them. Figure 15.

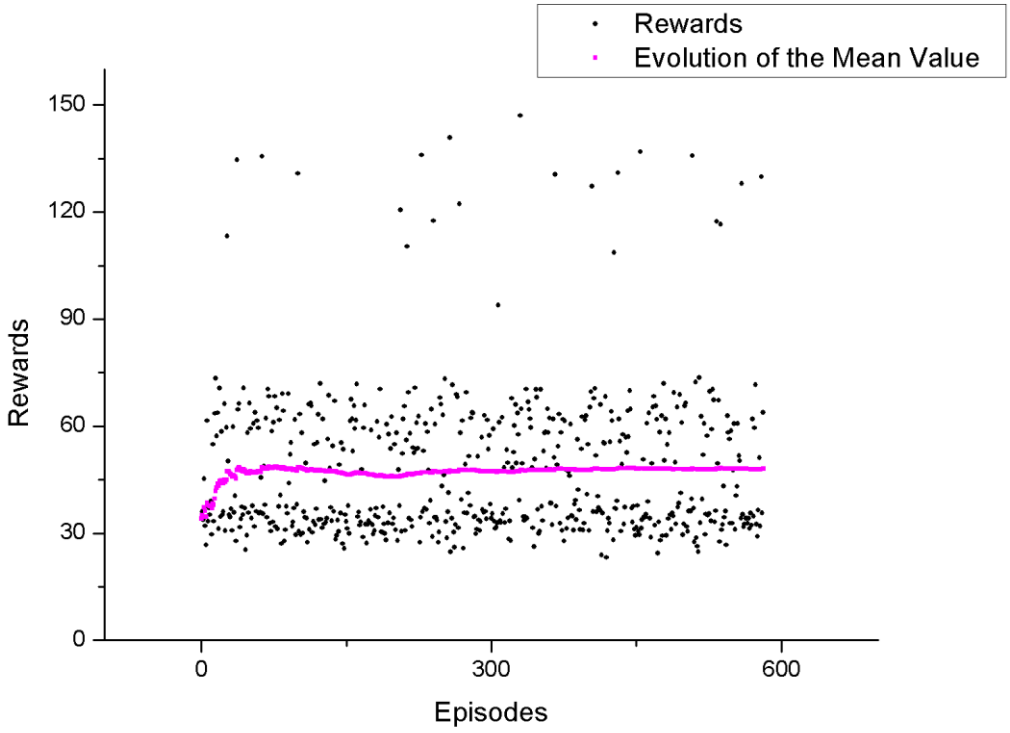


Figure 15 - Galaga results: Q-learning

On the other hand, the function approximation agent seems to have performed a little better this time, the mean value is slightly bent upwards, but that could be just noise. Still, better than slanted downwards. The problem is that again, Q-learning which seems to be mostly exploratory with so few episodes still bests our implementation of function approximation. Figure 16.

With full simulations the MCTS performs relatively well, however, its advancement seems really excruciatingly slow, given than full simulations on Galaga can last for minutes. Using short simulations limited to a handful of seconds accelerates the things a lot, and since we don't have a fuel depletion like in Galaxy Patrol, just giving the simulations enought frames to evade short-term death seems to work quite well.

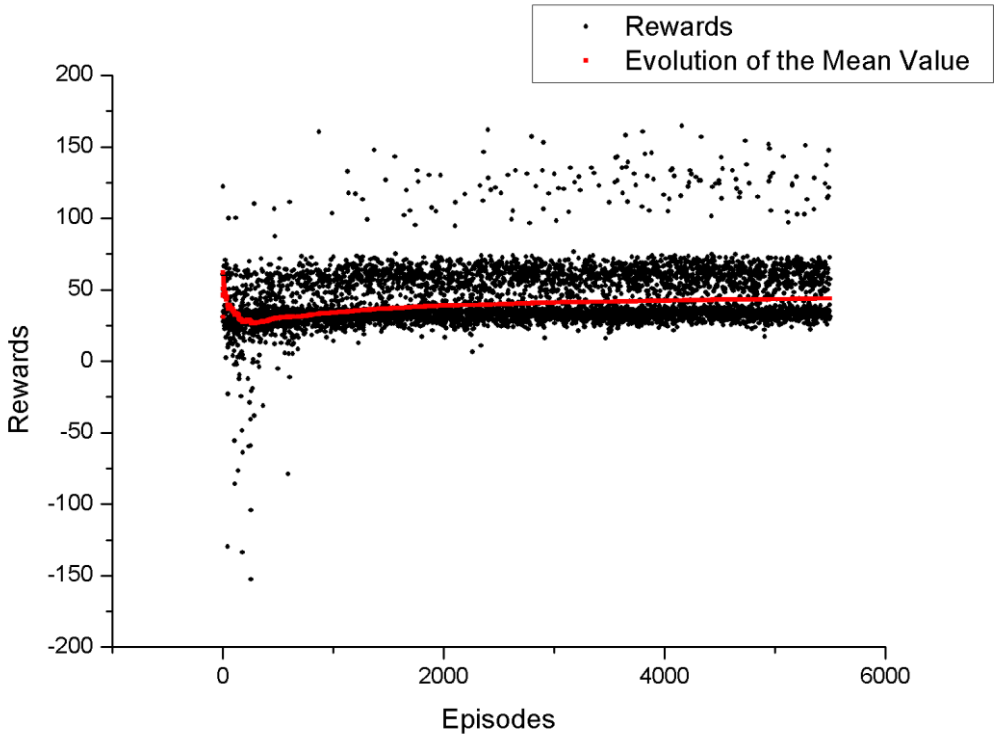


Figure 16 - Galaga results: Gradient Descent Function Approximation

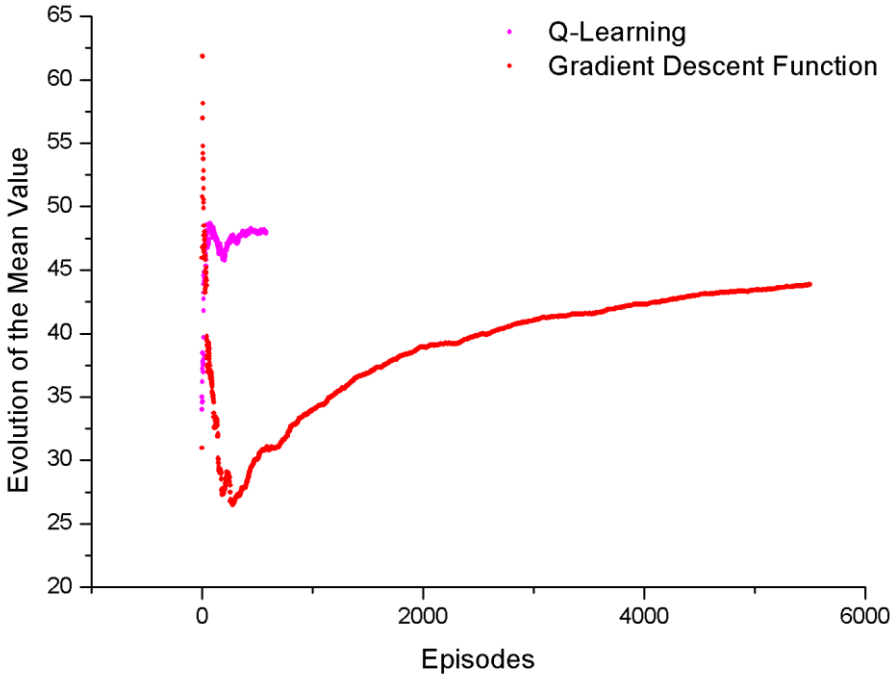


Figure 17 - Galaga results: Q-learning + Gradient Descent Function Approximation

5.3 Super Mario Bros

Super Mario Bros is the platformer par excellence. The player is supposed to reach the end of the level before the time runs out, grabbing powerups and evading or killing enemies by jumping on them.

Most buttons are used on mario, so we just removed the start and select buttons.

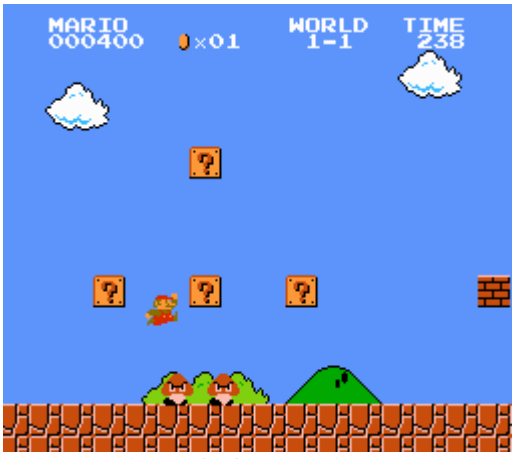


Figure 18 - Super Mario Bros

a) Reward and terminal functions

The agent is rewarded with +0.25 when the score goes up and +1 when he velocity of the player is positive and points to the right. (All levels of Super Mario bros are completed going to the right until the level ends). The agent is penalized with -1.5 when the character dies.

Episode ends whenever Mario dies, disregarding how many lives Mario has.

a) Results

The Q-learning Agent couldn't finish the expected episodes because the computer didn't had enough memory. This is because there were lots of opportunities to gain rewards, and each state with reward has to be saved as a key to retrieve the value.

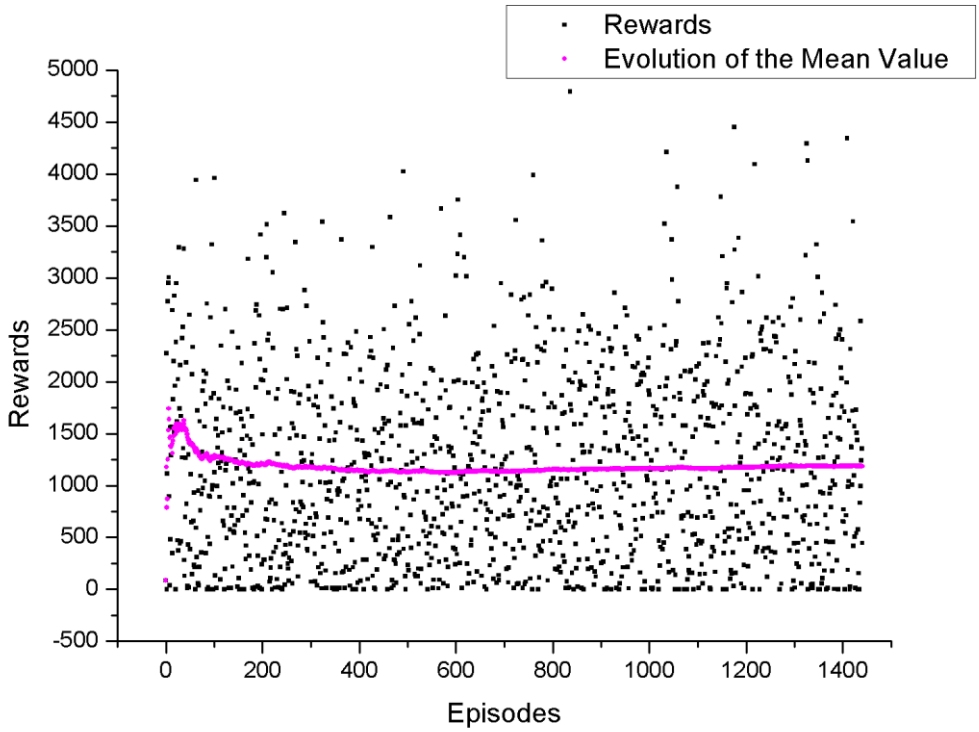


Figure 19 - Super Mario Bros results: Q-learning

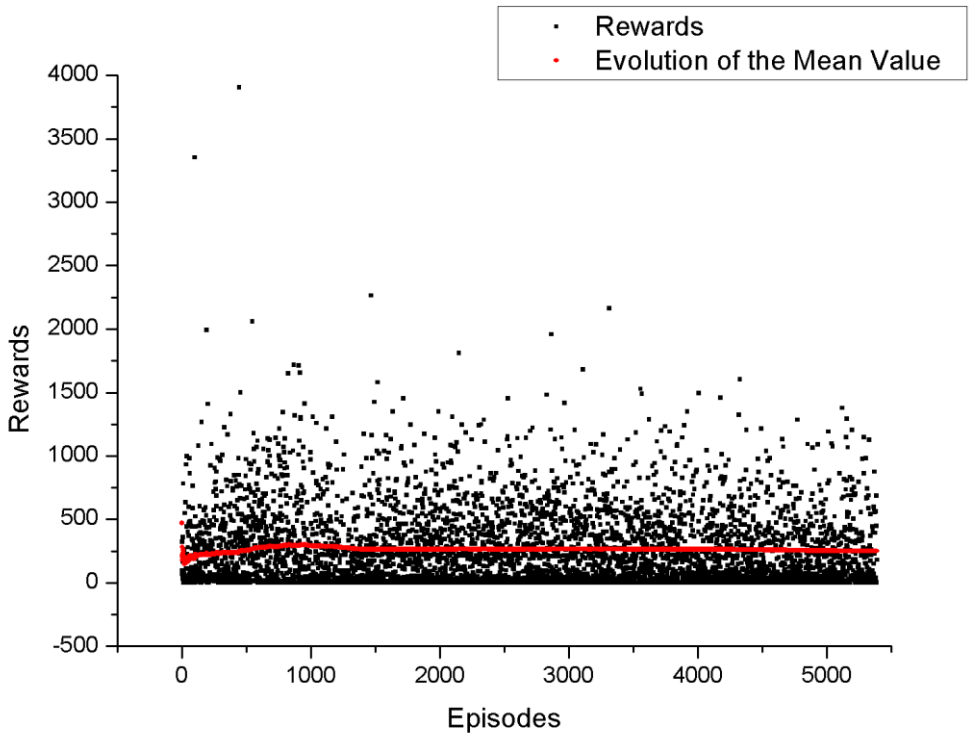


Figure 20 - Super Mario Bros results: Gradient Descent Function approximation

Again our Function Approximation agent performance was quite dissapointing, while the movements in game seem to be less random, this actually serves little its purpose since even the Q-learning initial exploratory efforts seem to perform quite better.

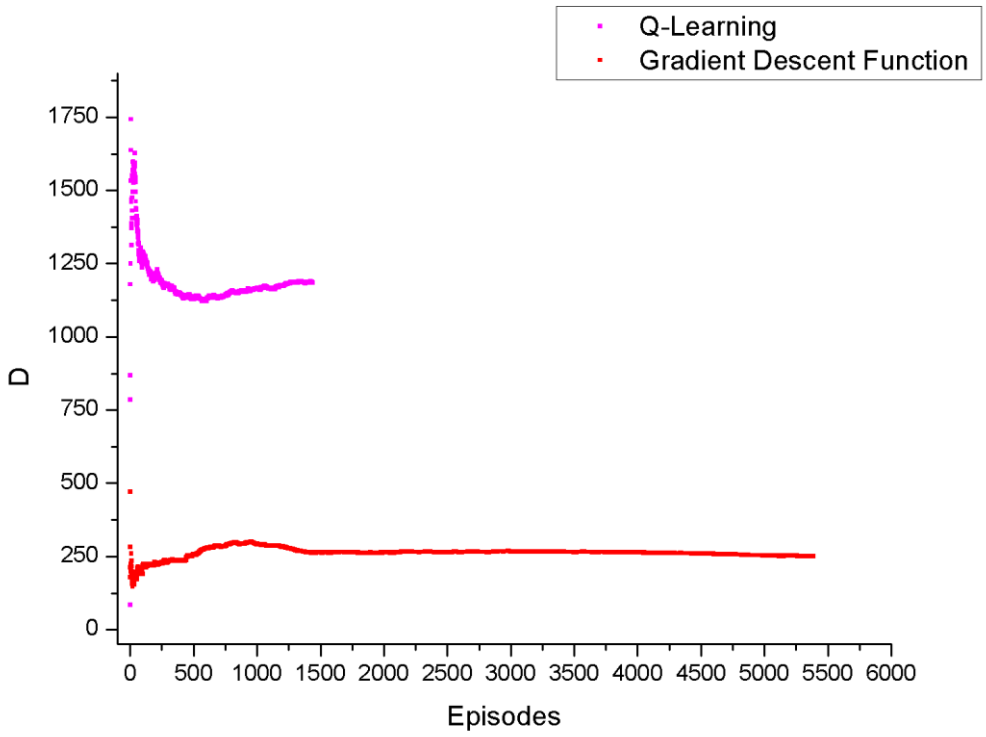


Figure 21 - Super Mario Bros results: Q-learning + Gradient Descent Function Approximation

6. CONCLUSION

In the case of Q-learning, it's obvious that saving the entire states as a key to retrieve the value is non permisible. For Super Mario Bros, a meager quantity of 1000 episodes required our computer to save several gigabytes of ram. More sensible ways to store or abstract and compare states should be implemented for this kind of agents to work on an environment such as this.

Another of the problems found is that for most NES games, there are addresses that count time played. This is distressing, because for agents that are unable to relate similar states, two states that practically differ in 1 bit would pass as completely different. This not only applies to time, but also the music (which might not be relevant in many games) is part of the state as well. For agents that are unable to relate states, devising a way to suppress these values. In the case of the music is easy, since as we have explained before, NES memory is organized in blocks that usually store the same kind of data in all its bytes. A block that has music only will store music, so if we consider the music not important we could just schew 256 bytes from the state.
For agents that are able to relate similar states, reducing the noise that those nonessential bytes generate might be positive as well.

For the function approximation agent, all we can see is that random exploratory movements best it. We could try to give even more episodes and see if the agent ultimately learns something useful, however, we believe the problem has been that the features choosen might have not been chosen correctly.

For the MCTS agent, there have not been substantial problems. The most problematic flaw was letting the emulator handle the savestates. Aside from making the protocol between agent and environment much more complex than it would need to be, it limits us to what the emulator can handle. For MTCS this is crucial since in scenarios where the winning scenario is far away from what the length of the search tree can initially reach, it can become impossible to beat the game.

BIBLIOGRAPHY

[1] "Sutton & Barto Book: Reinforcement Learning: An Introduction." [Online]. Available: http://webdocs.cs.ualberta.ca/~sutton/book/the-book.html. [Accessed: 05-Jun-2015].

[2] "Monte Carlo Tree Search." [Online]. Available: http://mcts.ai/index.html. [Accessed: 05-Jun-2015].

[3] G. Staff, "Flashback NES - GameSpot." [Online]. Available: http://www.gamespot.com/articles/flashback-nes/1100-6144735/. [Accessed: 08-Jun-2015].

[4] "Mario AI Championship 2012." [Online]. Available: http://www.marioai.org/. [Accessed: 08-Jun-2015].

[5] "FCEUX." [Online]. Available: http://www.fceux.com/web/home.html. [Accessed: 07-Jun-2015].

[6] "Romhacking.net - Homebrew." [Online]. Available: http://www.romhacking.net/?page=homebrew&platform=1&category=1&perpage=20&title=&author=&search=Go. [Accessed: 09-Jun-2015].

[7] "Nesdev wiki." [Online]. Available: http://wiki.nesdev.com/w/index.php/Nesdev_Wiki. [Accessed: 09-Jun-2015].

[8] "NES Hacker Wiki." [Online]. Available: http://www.thealmightyguru.com/Games/Hacking/Wiki/index.php?title=Main_Page. [Accessed: 09-Jun-2015].

[9] Adelikat et al, "FCEUX Help menu." [Online]. Available: http://www.fceux.com/web/help/fceux.html.

[10] "6.5 Q-Learning: Off-Policy TD Control." [Online]. Available: http://webdocs.cs.ualberta.ca/~sutton/book/ebook/node65.html. [Accessed: 10-Jun-2015].

[11] B. Tanner and A. White, "RL-Glue : Language-Independent Software for Reinforcement-Learning Experiments," *J. Mach. Learn. Res.*, vol. 10, pp. 2133–2136, Sep. 2009.

[12] T. M. VII, "The First Level of Super Mario Bros. is Easy with Lexicographic," 2013.

[13] M. ~G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The Arcade Learning Environment: An Evaluation Platform for General Agents," *J. Artif. Intell. Res.*, vol. 47, pp. 253–279, 2013.

[14] Adelikat, “NES Mapping.” [Online]. Available: http://www.fceux.com/web/help/NESRAMMappingFindingValues.html.

[15] “8.4 Control with Function Approximation.” [Online]. Available: http://webdocs.cs.ualberta.ca/~sutton/book/ebook/node89.html. [Accessed: 10-Jun-2015].

[16] “Monte Carlo Tree Search.” [Online]. Available: http://mcts.ai/about/index.html. [Accessed: 11-Jun-2015].