Reinforcement Learning in Videogames

Àlex Osés Laza

Final Project

Director: Javier Béjar Alonso

FIB • UPC

20 de junio de 2017

Acknowledgements

First I would like and appreciate the work and effort that my director, **Javier Béjar**, has put into me. Either solving a ton of questions that I had or guiding me throughout the whole project, he has helped me a lot to make a good final project.

I would also love to thank my family and friends, that supported me throughout the entirety of the career and specially during this project.

Abstract (English)

While there are still a lot of projects and papers focused on: given a game, discover and measure which is the best algorithm for it, I decided to twist things around and decided to focus on two algorithms and its parameters be able to tell which games will be best approachable with it.

To do this, I will be implementing both algorithms `Q-Learning` and `SARSA`, helping myself with `Neural Networks` to be able to represent the vast state space that the games have. The idea is to implement the algorithms as general as possible.This way in case someone wanted to use my algorithms for their game, it would take the less amount of time possible to adapt the game for the algorithm.

I will be using some games that are used to make `Artificial Intelligence` competitions so I have a base to work with, having more time to focus on the actual algorithm implementation and results comparison.

Abstract (Català)

Mentre ja existeixen molts projectes i estudis centrats en: donat un joc, descobrir i mesurar quin es el millor algoritme per aquell joc, he decidit donar-li la volta i centrar-me en donat dos algorismes i els seus paràmetres, ser capaç de trobar quin tipus de jocs es beneficien més de la configuració donada.

Per fer aixo, implementaré els dos algorismes `Q-Learning` i `SARSA`, ajudant-me de les `Xarxes Neuronals` per a ser capaç de representar la gran quantitat de possibles estats a la que m'afrontaré. La idea es fer una implementació el mes general possible. D'aquesta manera, si algú agafés els meus algorismes per els seus jocs, el temps que hauria d'invertir per adaptar el joc per l'algoritme sigui mínim.

Utilitzaré alguns jocs que s'utilitzen per fer competicions de `Intel·ligència Artificial`, aixi tindré una base a la que treballar i podré invertir mes temps a la implementació de l'algoritme i a comparar els resultats.

Abstract (Castellano)

Mientras ya existen muchos proyectos y estudios centrados en: dado un juego, descubrir y medir qué algoritmo se adapta mejor, he decidido darle un giro y centrarme en: dados dos algoritmos y sus parámetros, ser capaz de decir que juegos irán mejor con dicha configuración.

Para ello, implementaré los dos algoritmos `Q-Learning` y `SARSA`, ayudándome de `Redes Neuronales` para ser capaz de representar la vasta cantidad de posibles estados posibles del juego. La idea es hacer una implementación tan general como sea posible. De esta manera cualquiera que quería usar mis algoritmos para su juego, el tiempo que tendría que invertir para adaptar el juego al algoritmo sea mínimo.

Utilizaré juegos que se utilizan para hacer competiciones de `Inteligencia Artificial`, de esta forma tendré una base en la que trabajar y podré centrarme más en la implementación de los algoritmos y la comparación de resultados.

Índice general

Capítulo 1

Introduction

1.1. Project introduction

`Reinforcement Learning` is an area of `Machine Learning` in which we don't need to `supervise` our `agent` to teach him how to perform a `task`. In this case, the designed `task` will be to learn how to play decently some `video games` with specific implementations for each one of them. In order to achieve this task, the `agent` needs to be able to make an observation about the `environment`, in our case the `video game state`, and take an `action` based on how much `reward` thinks is gonna get based on its `training` and `experience`.

Nowadays, in the ever-growing `Video game Industry`, we still find `Artificial Intelligence` that feels exactly as the name suggests; Artificial. This is due to many factors. One of them is that `Computational Time` for the agents is really small, focusing more time on `graphics` rather than on `Intelligence` of the `NPCs`[1]. With this kind of techniques though, we apparently could get `agents` to perform decisions more `human-like` without the necessity of spending a lot of time `computing` their next action, it is even possible to train them `in-game`, learning from how the user plays the `game`, `adapting` to it.

1.2. Scope and Stakeholders

1.2.1. Stakeholders

In this section there's a description of the different people that might be interested in this project or might get some benefit out of it.

1.2.1.1. Developers

Just like me, many young and senior developers are becoming more attracted to `Machine Learning` and `Reinforcement Learning` because of the versatility and the potential of their `techniques`. This project will help whoever developer that reads it to understand about `Reinforcement Learning` and hopefully understand which problems can be more efficiently solved with which `Algorithm` and `parameters`.

[1]NPC stands for Non-Playable Character

1.2.1.2. Researches

This project might become a good starting point for researches that are interested in `Reinforcement Learning`. Also, it will give them a lot of data from different `algorithms` and `parameters` about their efficiency and results, making it easier to add new metrics and even extend the `algorithm` pool and compare my `algorithms` with many others.

1.2.2. Project scope

1.2.2.1. Objectives

The goal of this project is to successfully implement two of the most popular `Reinforcement Learning` algorithms: **SARSA** and **Q-Learning** to later compare the performance in three different games: **FightingGameICE**, **Mario** and **Tetris**.

Once we have these implementations and we can run the algorithm for the different games, we will be aiming to try and find a correlation between the algorithm, the parameters and if it is possible to give a guideline on when one algorithm, with certain parameters, is better than the other depending on several factors such as:

- Consistency of the agents
- Quality of the agents

Another objective of this project is to proof if it is possible to create an agent that decently plays a game without having to worry all too much about how the game's mechanics work with a relatively low amount of training.

1.2.2.2. Possible Difficulties

1.2.2.2.0.1 Computational Time

The `Algorithms` we'll be taking into account in this project are all **really** expensive timewise. Not only that but they are filled with a seemly infinite amount of parameters that might change it's behaviour and, therefore the `algorithm` may become better or worse depending on them. This means that we will have to execute **each `algorithm` numerous** times tweaking parameters to see which one fits best for our problem. To solve this problem, parallelization is **vital** and many possibilities will be considered throughout the project to see which is the best approach to this problem, will there be just enough with 2-3 machine or, on the other hand, third-party cloud servers will be necessary for such computing volume, we'll have to see until we do some tests for the given problems.

1.2.2.2.0.2 Memory

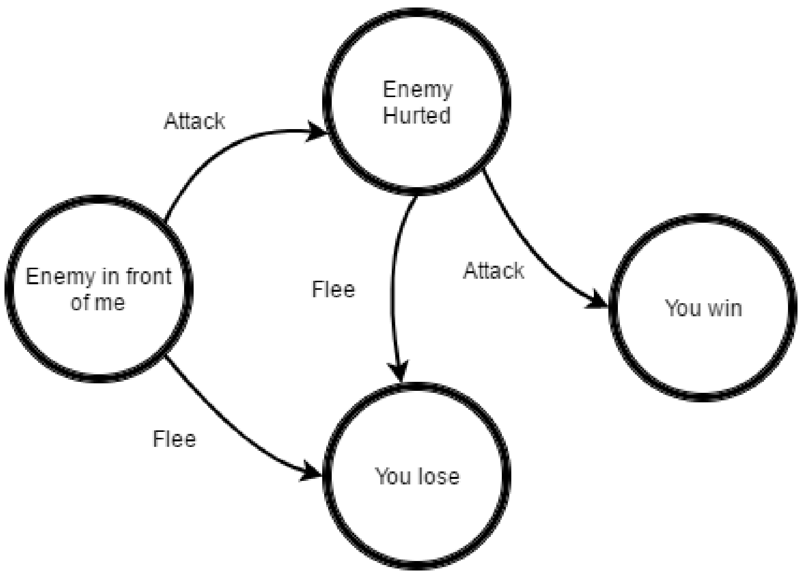
Our problems will have a **huge** state space if we don't represent their `environment` accordingly. This means that we have to be `really` careful with our `Game State` representations of the different games because if it is too large, it will take way too much memory and the problem will become `unviable` to compute. To solve this major threat, we need to think about an smart representation of the given `environment` and avoid using `algorithms` implementations that are based on tables rather than on `Neural Networks` or `Function approximations` so we reduce as much as possible the memory usage.

1.3. State of the art

1.3.1. Reinforcement Learning

As I mentioned earlier, `Reinforcement Learning` is described as an area in `Machine Learning` that pretends to know which actions an agent must perform in his environment in order to maximize the number of cumulative reward he will get during its execution. This area could be considered to be a form of `unsupervised learning` from machine learning, because we are not constantly monitoring if the agent is behaving correctly or not rather than giving our agent tools to make him learn and understand which kind of actions in which kind of scenarios will give him the most reward in the long run. This is a vast topic, therefore I will only try explaining the little bits that are interesting and useful for this concrete project, for more information about this area I highly recommend reading the book in the bibliography: *Sutton and Barto Book: Reinforcement Learning: An introduction*, which is the book that inspired and taught me all the concepts that will be explained in this section and throughout the document.

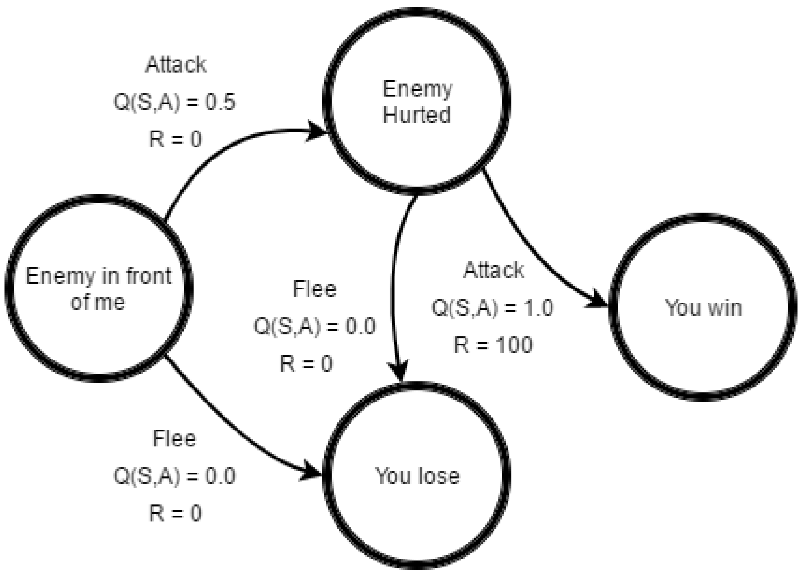
This area pretended to give a solution for **Markov Decision processes (MDP)**. An `MDP` provides a mathematical framework for modeling decision making environments where outcomes are partly random partly in control of the agent. This is actually the perfect way to actually represent the workflow of a game:



As we can see, our states would be the current `Game State` we are in and the arrows would be the possible action we would take. This way we can easily represent something like: Ïf I am in state S and I perform the action A in which State S' should I end up to?". Now what's left is a way for our agent to know how good being in a particular state is. To solve this we need to introduce the concept of **Value Functions**.

1.3.1.1. Value Functions

A `Value Function` is a simple way to represent how good is to be in a state. This can be done in many many ways but, the one that we will be using for this project and I felt was most interesting was to use the **State-action Value Function**. What this `Value Function` represents is the reward that we would have if we were in a state S and we performed an action A, represented as $Q(S, A)$.



Now with this information, we can already see the logic of the agent taking form. As we can see in the example, all those pairs of state-actions that we can perform that, at the end, leads us to the final reward are higher than the ones that lead nowhere. Therefore, our agent will decide to take those actions in those particular states since he thinks that it will get the most reward by doing so.

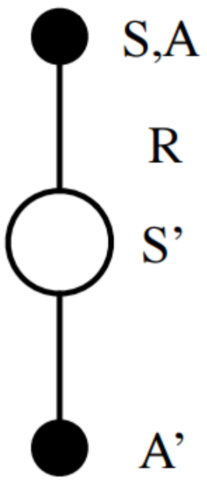
And now comes the interesting part. For now we have assumed that we already knew the values of the `Value Function` but, if we already had, there wouldn't be a problem to solve, neither would this project exist. Therefore, we need to find an efficient way to calculate those pairs of state-actions $Q(S, A)$ and this is what `Reinforcement Learning` is all about. In this project we'll be comparing two of the main, very similar, algorithms from reinforcement learning. Note that both converge to the optimal solution at the end, proof of this can be found in the book refered before.

1.3.1.2. SARSA

The name of this algorithm is actually the abbreviation of what this algorithm is based around. `SARSA` stands for:

- **S**tate
- **A**ction
- **R**eward
- **S**tate'
- **A**ction'

An this is the simpliest way to express what this algorithm actually does. We receive a pair State-action, which give us a reward, ending up into an state S' and deciding to perform an action A' for the next iteration. This is better seen in the following graph:



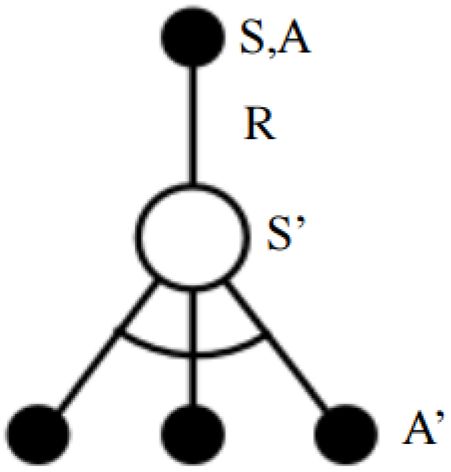
With this kind of workflow from the algorithm, what we achieve is simple way to update our value function $Q(S, A)$ in order to propagate the rewards that we get for each pair of state-actions, enabling the agent to know before hand which actions are more likely to be beneficial for him at each state. The updates on the `Value Function` looks something like this:

$$Q(S, A) = Q(S, A) + \alpha(R + \gamma Q(S', A') - Q(S, A)) \tag{1.1}$$

With this formula we're left with a simple way of updating our `State-Action value function` based on what reward we do receive now, and what potential reward we will get being at the pair state-action S'-A'. Also, note that we have two constants that need to be initialized by the programmer and it will range between **`0`** and **`1`**. The parameterα is referred to as the `Learning rate`, and it indicates the `rate` in which the agent should forget what he has learnt and focus more on the current reward and the future reward. Similarly, γ represents how much the agent cares for future rewards, since it strengthens the value of the `value function` in the state we will end up to rather than on itself or the immediate reward.

1.3.1.3. Q-Learning

The main idea of this algorithm is pretty much the same as in `SARSA` but with a little twist. `SARSA` is considered to be an `On-policy` algorithm since we are checking values that are being selected in the current execution. Now, with `Q-learning` we are `Off-policy` which means that we are not necessary looking at values from the current execution, enabling us to be much more aware of the surrounding states and their `value functions`. The workflow it follows is something like this:



As we can see on the graph, we are now receiving a pair of State-action, which gives us some sort of immediate reward, which leads us to another state S'. Until now everything looks the same as in `SARSA` but, here's where `Q-learning` proposal comes into play. Instead of just taking a decided actions as our A', we are considering among **ALL** of the possible pairs state-action. Making the agent much more aware of his possibilities and therefore, in theory, a bit smarter and faster to train. This is how the update function should look like for a `Q-Learning` algorithm:

$$Q(S, A) = Q(S, A) + \alpha\big(R + \gamma \max_{a'} Q(S', a') - Q(S, A)\big) \tag{1.2}$$

As we can see, now we have an update function very similar to `SARSA` but that now takes the maximum `State-action value function` that is available to us in the state we end up to. This should make the agent learn faster which is the optimum way to move around the environment, at least in theory. Once more we have the two same parameters as before which meaning remains the same in both of the algorithms.

1.3.1.4. Policies

So far we have: a good way to represent the behaviour of our game and agent, a way to know if an action is better than another, and two fantastic algorithms that will compute for us which states are better to be than the others. What's left then is for the agent to know **which** actions should perform given an state. This might seem simpler than what it really is.

The first intuition would be to just greedily select our actions. That way we would **ALWAYS** be moving to what the agent believes is the best possible choice. This is a really bad idea when we are trying to **train** our agent since, if we are choosing the actions that will give us the more reward the chances are that we will be missing states that will give us a lot more rewards simply because we didn't even know they are there. To solve this, we need to introduce some **explorative** steps to our agent, making him try out new things from time to time to see if there's any better way to explore his environment. The solution has a name: $\epsilon - greedy$ policy.

What we achieve with this policy is a `pseudo random` action selector, which depending on the value of

ϵ the agent will decide if he will take a greedy action or a random one therefore, exploring. This ϵ value will have to be decreased every time we visit an state, making our agent decide more randomly at the beginning or at the end of the episode, depending on what we are interested the most.

1.3.1.5. Function approximators

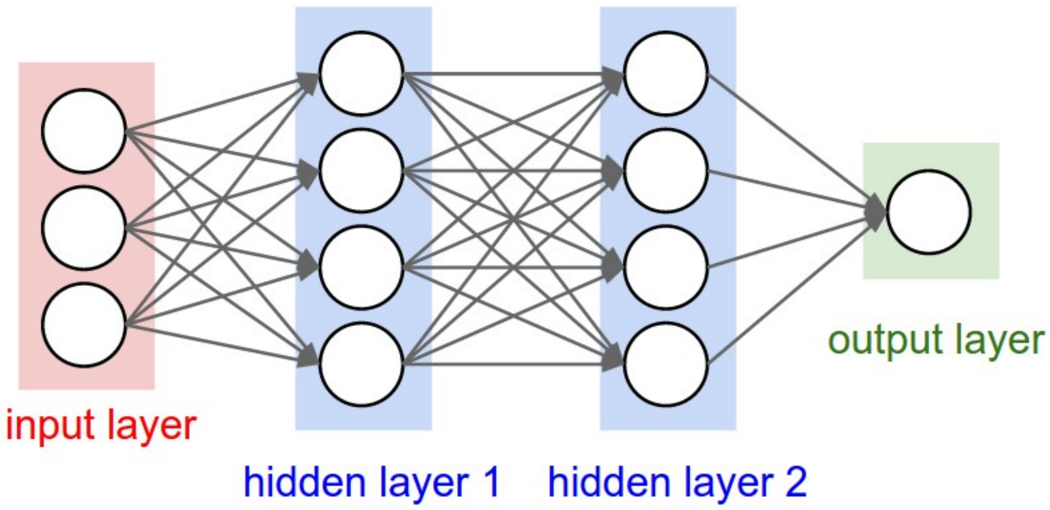
Until now we asumed that the problem we are trying to solve via `Reinforcement Learning`, a video game in this case, was just small enough to perfectly fit in memory. Well, truth is that all interesting games can't actually fit all their possible states in memory making this strategy pretty much useless. Unless, of course, we use some kind of function approximator that help us represent such big state space in a much smaller form. Just to give you some numbers, an average game, for example the game "Go"which `Google` is trying to create a good agent for it, has around $3{,}72 \times 10^{79}$ different states and this doesn't even take into account the number of possible actions for each state which would lead us to store a matrix of size $3{,}72^{79} \times numberOfPossibleActions$. Now that it's been clarified why wee need to these approximators let's go through some of the possible solutions to this problem.

There are mainly two kinds of approximators: Linear Approximation and `Neural Networks`. For this project we'll be using the second one, since it gives the possibility of learning non-linear value functions, which is the kind of value functions games do have. The main idea is that instead of looking up on a table the value that the state would have, we will be consulting the `Neural Network` to get the value for us. Of course this gives another level of difficulty to the problem, since the values we'll be getting will not always exactly be what they should, but for some reason they're called function approximators. Also, since we are using `Neural Networks` to solve this problem, we will have to deal with problems and limitations that `Neural Networks` have. This is why there's a whole new topic within `Reinforcement Learning` that specializes in this kind of configurations. It is called `Deep Reinforcement Learning`.

Neural Networks

`Neural Networks` are a really powerfull tool that can be understood quite easily if we visualize it as a black box where we have some inputs and, some how some way, the result in which the neural network has been trained for appears as an output. When we are trying to make use of them for our application though, we need to have a deeper understanding on how they work and how they manage to get this kind of behaviour so we can efficiently train them by tweaking some of its configuration.

This represents the basic structure of most of the `Neural Network` uses.



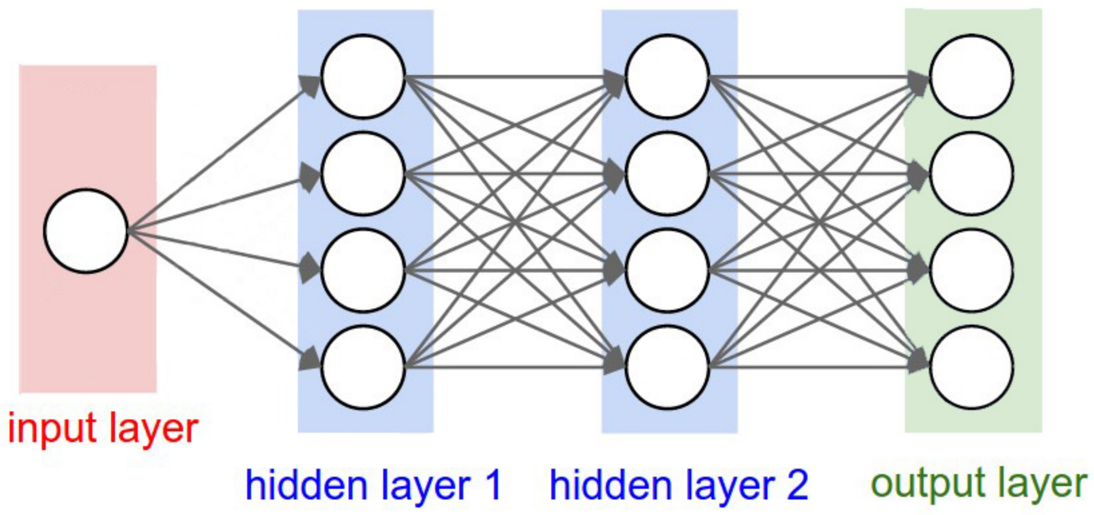
As we can see, we have a series of input nodes which will be used as an entry point to the neural network. Then, we have a number of hidden layers that have a number of nodes and, finally we have one or more nodes at the output layer with the result. Note that all of this is configurable, we can add or delete nodes as we wish depending on the necessities of our problem. The more nodes and the more layers, the more complex functions our `Neural Network` will be able to represent.

This are not the only parameters that can be configurated though, there are **A LOT** of possible parameters that we can tweak to make the neural network adapt to the kind of problem we are solving. Things like: Function activation, learning rate, weight init... Note that we listed a parameter called **learning rate**, this parameter is the one that our algorithms refer as α.

What Neural Networks do on the inside is to give a weight to each connection (represented as an arrow in our image) so that, at the end we have our desired result for the input values we've given. This is done by applying an **Stochastic Gradient Descent** which basically means that, each time we feed our neural network with a pair of input and output to train it, internally it will be changing those weights, using that technique, minimizing them to better fit the pair we've provided.

At the end of the training, we should have an optimum weight distribution which will represent the function that we are looking for. One disadvantage for the use of this kind of approximator is that its configuration is often based on trial and error, having nothing more than vague intuitions on what to change based on previous experience.

There are two ways of using `Neural Networks` for reinforcement learning, either you feed the network with a pair State-Action and get its value or, the one that I chose:



As we can see, we give as an input the state we are currently in and, the neural network, will return, for every posible action the value function for that pair State-Action. I chose this configuration because when using `Q-Learning` it will take just one call to the neural network to get the desired values, instead of the number of actions, making the algorithm that much more efficient.

1.3.2. Deep Reinforcement Learning

In order to make `Neural Networks` actually work in `Reinforcement Learning` we need to battle 2 of its main flaws:

- **Correlated values** often gives trouble to neural networks.
- **Constant Updating** values from the state they were before can give a really hard time to our neural network, making it not learn properly.

Luckily, this two problems have a kind of simple solution that will work for both our algorithms.

Solving Correlation

To avoid giving the `Neural Network` too many correlated on time values, we will be training the neural network by batches. This means that we will need to store the information of the whole execution and then take a subset and shuffle it. This way we are decorralating the values, making it much easier for the `Neural Network to learn`. How we save the information will be discussed a bit further into the document, when we talk about**transitions**.

Solving Constant Updating

Since `Reinforcement Learning` is all about getting a previous value from an state and updated in the direction of what it should be, since this can become a problem for our `Neural Network`, what we are going to do is to actually have **2** `Neural Networks`. One will be used just as a çonsulting one.ªnd the other one will be the one that its going to be updated. This way we avoid constantly consulting and updating the `same` one, having to swap which is the consulting network and which one is the other. Here's an example of how the function updater should look like for SARSA, where Q represents the value function that's being updated and Q' represents the frozen neural network (the consulting one).

$$Q(S,A) = Q'(S,A) + \alpha(R + \gamma Q'(S',A') - Q'(S,A)) \tag{1.3}$$

Transitions

To do this kind of batch training we need some way to store the interactions of our agent and its environment. To do this we are going to use `transitions`. A transition will represent an State S, the action A that was taken, the reward R that we got from that pair State-Action and in which state S' we ended up to. This way we could actually reproduce the execution extracting and calculating the value functions on a single go, instead of having to waste time during execution at every step.

1.3.3. Sources of Information

There's a lot of information and publications about this topic, and not only for `Video games`. Even in the `UPC` there has been some projects that also addresses the `Reinforcement Learning` topic for example *David Bigas's TFG in June 2015.*

There's also a good amount of papers that work around the idea of comparing `Reinforcement Learning` algorithms,for example Vaibhav Mohan's paper[2]explores some algorithms and compares them for an specific game, in this case: racetrack problems. Most of these kind of papers work around the idea of having a problem, which is the best way to solve it using `Reinforcement Learning` techniques. This project, on the other hand, I pretend to give it a twist and, given an algorithm, tell for which kind of problems it would work best and with which parameters we can get the most out of it.

In this project all `Algorithms` will be implemented by me, the reason being that I want to make sure I have full control over the `Algorithms` and avoid calling one `Algorithm` better than another one just because I picked one that had a much more efficient implementation than the other, making it seem it is the best possible solution when it is actually not. This is the main reason why I don't want to start my project from any other one.

This project will mostly be based in the knowledge of the book*Sutton Barto Book: Reinforcement Learning: An Introduction*, which is considered to be the best book in this subject not only for the concepts it introduces but for the way it explains the key concepts of `Reinforcement Learning` which provides a really intuitive way to understand ideas that are really abstract and complex. This book is currently being re-written for the release of its second edition, the book is called

[2]Link to the full paper: paper:https://pdfs.semanticscholar.org/4529/71237b93968b89e7d57b62bc40485331b047.pdf

Capítulo 2

Planning

2.0.1. Methodology

I will be following a sequential methodology for the development of this project, adapting some of the fundamentals of `agile methodologies` such as frequent meetings with my project tutor so we can discuss how the project is going and how should it continue. I will structure the project in these phases, taking into account that if I am in a phase, is doesn't mean I will be exclusively doing that objective rather than focusing most of my efforts on it.

2.0.1.1. Collecting Information

This is probably the most important phase since getting good information about the topic is `vital` to understand how to correctly make efficient and working algorithms. I will be spending **a lot** of time in this phase, making sure I've got all the tools I need to understand and implement the algorithms for each game. This phase will take place in 8 weeks.

2.0.1.2. First Game: The Snake

Since this is the first game I will be implementing both algorithms, is will take me some more time than usual since, I don't only have to represent the states of the game for the algorithm but I also have to implement it. This phase will take me around 3 weeks in which I will have to implement and test these algorithms:

- Q-learning
- SARSA

2.0.1.3. Second Game: Mario Bros.

For this phase, I will already have implemented and tested both algorithms but I will need to represent the `game state` and adapt it so the algorithms work. This is meant to take me 2 weeks. Once more, is will be done for the following algorithms:

- Q-learning
- SARSA

2.0.1.4. Third Game: Fighting Game

Similar to the previous phase I will be representing the `game state` once more for this game. This is also meant to take me 2 weeks, one week for each algorithm:

- Q-learning
- SARSA

2.0.1.5. Documentation and Training

For the rest of the time schedule I will be focusing a lot on making the best documentation I can possibly produce and preparing the presentation. In the meanwhile the algorithms will be left training for the 3 games. It will also be in this phase where I will be plotting and interpreting the results. This phase is intended to take the last 2-3 weeks until the delivery of this document.

2.0.2. Schedule

2.0.2.1. Estimated project duration

This project will be made in approximately 4 and a half months, starting on 1st of February and ending on the due date, some time around the 20th of June.

2.0.2.2. Considerations

Keep in mind that this planning could be revised and updated depending on the evolution of the project. Thanks to using adapted agile methods this should be no problem and will not affect to the main core of the project.

2.0.3. Stages

These will be the steps that I will follow sequentially to make of this project, a reality. Stages 6.2.3 and 6.2.4 are meant to be part of an iteration, and can be reproduced several times in the length of this project to add new Games to compare the algorithms with.

2.0.3.1. Information Retrieval

This is the core of the project, It's `vital` to find good source of information to really learn how to apply the theory to the reality. I'll be looking at two main topics of information.

- Reinforcement Learning
- Neural Networks

The first one is pretty obvious, if I am going to produce good algorithms based on Reinforcement Learning firs I will need to know the basics of this area. I will be looking for whatever source of information I can get, videos, papers, tutorials... On the other hand, the second one might not look too related to the project's topic but it actually is. `Reinforcement Learning` algorithms are very memory heavy, and I might not be able to represent all states and all actions together in a single matrix or another data structure, that is why I'll be looking into the viability of `Neural Networks` to see if I can use them in my advantage.

2.0.3.2. Algorithm Implementation

From everything I've learnt from the previous stage, I will be implementing the algorithms using a simple game with a reduced number of states and actions so I can test them properly. This stage wouldn't take long if the objective was to just make them work. However, I want to make sure I can implement them as generalized as possible so I can save some time in future stages and make it easier to add new games the agent will be able to play.

2.0.3.2.0.1 Related Tasks

This stage will have two main smaller task:

- **SARSA Algorithm Implementation**
- **Q-learning Algorithm Implementation**

The estimated time is very similar for both of them, having to put a little less effort for Q-learning `implementation` since it's a variation of SARSA and all the experience gained by implementing SARSA will be of `huge` help.

2.0.4. Game State Adaptation

The complexity of this stage will heavily depend on the success of both the last two stages. If I managed to really learn the idea behind `Reinforcement Learning` and I have implemented general enough algorithms this stage should just be to adapt the `Game Environment` into the set of `Game States` and `Actions` the algorithms need to properly work.

2.0.5. Agent Training

Once I have the `Agent` properly working, it will be time to start its training. The time on this stage may vary depending on the complexity of the game: the more `States` and `Actions` the longer the `Agent` will need to practice. Luckily, this process doesn't have to be monitored that much and I will be able focus on another tasks.

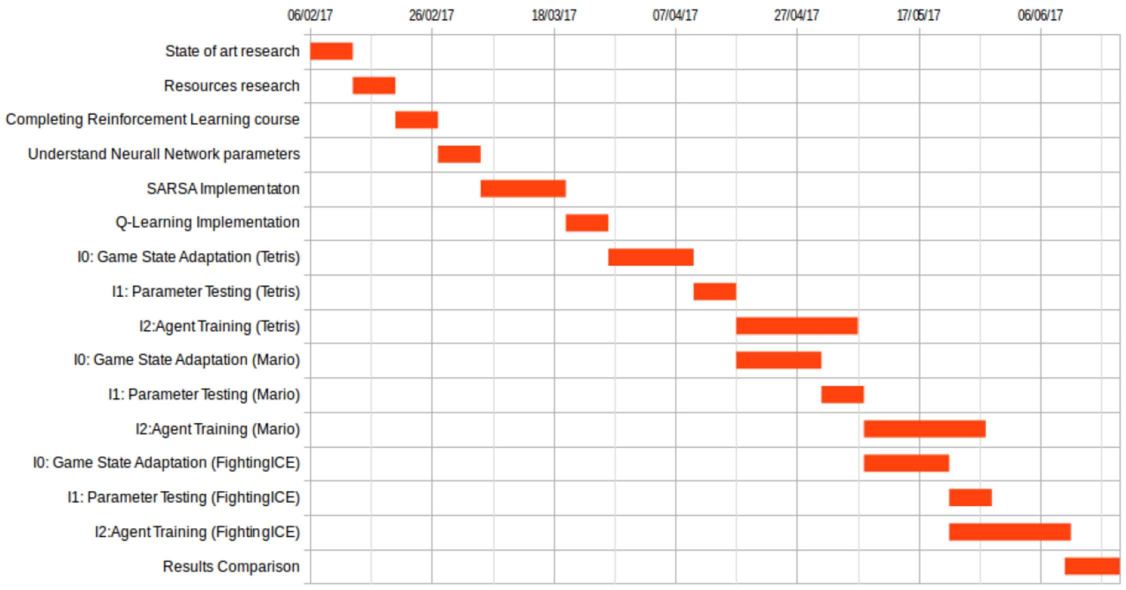
2.0.6. Documentation

In this stage I will have a lot of documentation already done but I will focus on improving it and mostly on comparing the different data gathered by the agents so I can compare the algorithms among themselves and among the games. The objective of this stage is to be able to show the results as clean and understanding as possible.

2.0.7. Estimated Time

Stage	Estimated Dedication
Information Retrieval	100 hours
Algorithms Implementation	20 hours
I0- Game State Implementation	35 hours / Iterations
I1- Agent Training	40 hours / Iterations
Documentation(including whole project)	100 hours
Total	220 + 75 * Iterations
Total (Objective: 3 games)	625 hours

2.0.8. Gantt



2.0.9. Action Plan

As stated earlier, I will be following an adapted agile methodology for this project. This means that this initial planification doesn't really have to be a mirror from the reality and can be adapted depending if whether or not I need more time. Also note that I will not be following a waterfall model methodology, this means that If I'm, for instance, at the first stage this doesn't mean I will be exclusively focusing on retrieving information rather than focusing more of my time in that direction. This also means that while I am at stages like, `Algorithm Implementation` I will be also documenting the algorithm and if that I am at `Game State Adaptation` for a game I will most likely already be taking data and training the previous Agent. In any case, if I need more time because I didn't expect something, I can always do less games to compare. On the other hand, if I find myself that I actually have more time than expected, I will be able to expand the game pool and make more agents for more games.

2.0.9.1. Possible Difficulties

For this project there are many tasks that can get out of hand really quickly, here I will denote those task from the gantt chart, and trying to expose what they would mean to the overall project.

2.0.9.1.0.1 I0: Game State Adaptations

In this task things could get really tricky. In this specific I need to be able to translate what is happening inside the game so the agent can give an action for the current state of the game. This task is probably the most important and complex, even more than the algorithm, and can take a lot less time than in the planification or a lot more. If there are difficulties to perform this task during the course of the project, 3 things can occur.

- **The Game gets Cancelled** If the game state is taking too long because of its unexpected complexity to process, the game might be cancelled and not taking into account for the experiments. This is not necessary too bad, since because the game getting cancelled, gives the chance to either, try and incorporate another game, which would be easier, to the project or give more time to the other parts of the project.
- **Another Following Game gets Cancelled** If the game state is taking too long but I decide that it is really important to have that game in the project, cancelling other not so interesting game will be considered. This will only be done if the implementation of the game state is going to take way more time than expected, forcing to not have time to implement other games.
- **The Testing Parameters Task gets shortened** If this task is going to take too long but the time is reasonable, I might just take some time from the I1:Parameter Testing task from the given game.

2.0.9.1.0.2 I1: Parameter Testing

At first glance, this task should be only time consuming the first time that it's done. I will try to do an script that automatically tests all possible parameters that make sense for the problem, probably helping myself with cloud servers. This means that rather than taking time from a programmer, it will just be like the training task, where I can just leave it computing and working on the next game. Of course, this is the theory and I might encounter some problems like the tests are not working properly or the computer shutting down before being able to write a result for the test, requiring me to invest more time either improving my tests or taking care of the computer. Two things might happen if this task gets delayed, which are exactly the same causes as stated in the previous section:

- **The Game gets Cancelled**
- **Another following Game gets Cancelled**

2.0.10. Change of plans

2.0.10.1. Context

In this project I will be comparing different reinforcement learning algorithms (Q-learning and SARSA) on different games (Tetris,Mario,FightingICE) for parameters of a certain range that will depend on the game. The computational power is a problem that I am and I will be having through this project and I will be using simplifications of what normally people uses (normally they usea the whole game screen as input).

2.0.10.2. Planification

There has been two major factors that forced me to change my planification. Both of them were expected on the original planing, what was not expected is how long they took to finally be completed forcing me to spend much more time than I was supposed to in those stages and forced me to make some decisions. The factors are:

- **Game State Adaptation**. The problem with this stage was that it is actually two steps in one, and I wasn't able to see the complexity of it at the begining of this project. This stage should be separated into two: `Understanding how the game works/sends information` and Game State Adaptation.
- **Parameter Testing**. This stage was supposed to have low development time and was supposed to allow me to focus on other stages while it was being done. This wasn't the case since all the plans I had to be able to compute and test parameters in an easy and highly parallelizable fashion were a big failure. This forces me to train all algorithms on my not so powerful laptop, meaning that my computational power is **greatly** reduced.

2.0.10.3. Methodology

Since I lost a lot of time, as I stated before, the methodology will be slightly changed. First of all, there are one new stage: `Understanding how the game works/sends information` which will take place before each `Game State Adaptation`. This is because the challenge is not only to adapt the game state and give it to my algorithms, but it is also a challenge to take a game that has been done by someone else and make sure that it's functional. That means making sure that it runs well on my computer, that I have all dependencies, that I am able to generated a package with all the dependencies necessary for the game to run with my agent, etc...

Also, It is possible that one game will be left out of the project since I might not have enough time to test it properly. Nevertheless, one of the games: The snake will be substituted by Tetris if there's enough time. The ones that will for sure make it are: Mario and FightingICE

2.0.10.4. Alternatives analysis

2.0.10.4.0.1 game low reliability

. There has been games that when I was trying to execute and make my neural network to learn, they became stuck for a variety of reasons. Forcing me to pay atention when the tests were running and killing the process if necessary.

To solve this, I made an script that checked all the time if java was stuck and, if so, it would kill the program so it could keep iterating through the episodes.

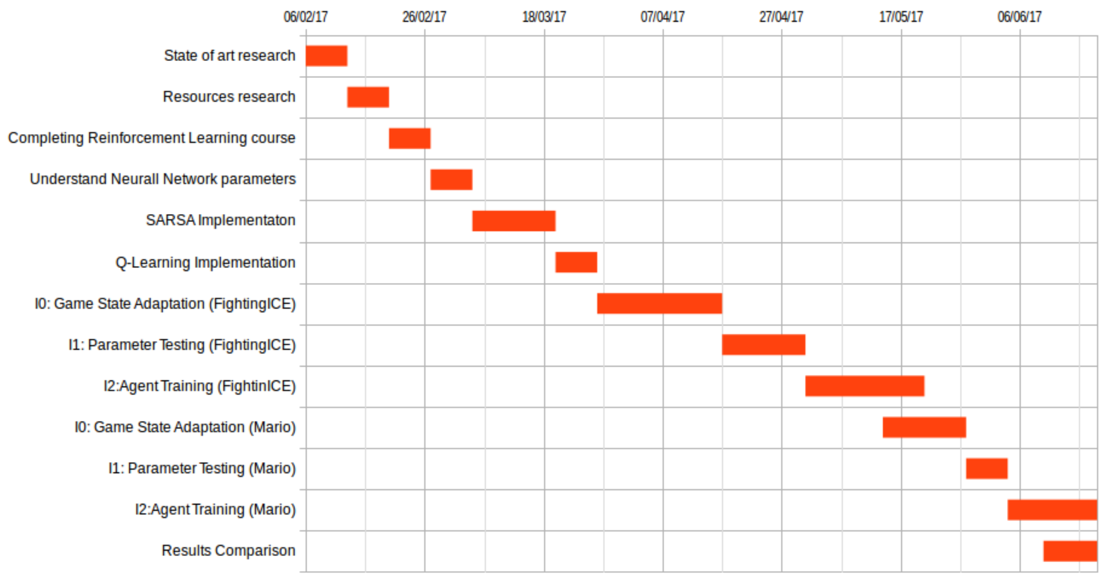
2.0.10.4.0.2 Computational power issues

At the end I wasn't able to use the i7s that I would want to so I decided trying to use Amazon's AWS services. However, this was a failure too since one of the games was not created with this idea of training neural networks. Even though they made patches so this possibility would be viable(for example: allowing the user to disable the game window), they were poorly executed and at the end of the day it was a source of problems and was not worth the time to try and make it work on amazon or on any other cloud service (it was not an specific problem of amazon rather than a general problem to compute the agent in a server).

Unfortunately, to solve this I need to narrow down the scope of the tests. Instead of trying all possible parameters that are available I will be greatly reducing it's range based on the game so I can realistically train the agent with different parameters.

2.0.10.5. New Gantt

As a result of the delays on my project, this would be a better approximation on how the planing went and how it is supposed to be from this point onwards. Note that I did not separate the `Game State Adaptation` into the two stages mentioned before because I thought it would be confusing and the change on the planning would be harder to undestand.



2.1. Budgets

2.1.1. Considerations

This budget will be constantly updated at the end of each iteration so we have an accurate representation of the budget spent at the end of this project. In any case, it will only affect to the amount of time spent in the project (Human Hours) since the rest of the requirements are pretty straight forward and unexpected spends on other resources will definitely not be likely. The causes for more human hours are simple: either we need more time to adapt the `Game State` for the algorithms or we need more training time for the `Agents`.

2.1.2. Human Resources Budget

This project will be developed by only one person (me). So I will have to take different roles. Here are the roles and it's medium salary in Spain:

Cuadro 2.1: Human Resources Budget

Role	Estimated Hours	Medium Salary per Hour	Total Estimated Cost
Project Manager	25	50 /hour	1.250
Software Developer Engineer	600	35 /hour	21.000
Total Estimated	625		22.250

2.1.3. Hardware Budget

Here is the software listed to test and implement the project, I will also include some optional hardware that will make the project go much faster in the testing part.

Cuadro 2.2: Hardware Budget

Product	Units	Estimated price	Useful Life	Total annual estimated amortization
LenovoG50	1	349,99	6 years	58,33
Intel i7-6700[1]	2	618	6 years	103
Total Estimated		967,99		161,33

[1]This hardware is optional and not necessary. It will help a lot to compute the trainings though.

2.1.4. Software and License Budget

Finally, these are the third-party software I will be using for the project.

Cuadro 2.3: Software Licensing Budget

Product	Units	Estimated price	Useful Life	Total annual estimated amortization
GitHub Premium Service[2]	1	7/month	6 months	42
Ubuntu 16.06	1	free		0
IntelliJ Community Edition IDE	1	free		0
Total Estimated		42		42

2.1.5. Total Budget

Adding all the budgets:

Cuadro 2.4: Total Budget

Concept	Estimated Cost
Human Resouces	22.250
Hardware	967,99
Software Licensing	42

2.2. Project Justifications

2.2.1. Project Objective

The main objective of this project is to understand and test different `Reinforcement Learning` techniques in order to demonstrate which are the best approaches to different `Video games` among all possibilities this area of `Machine Learning` has to offer.

Another objective is to identify the properties that make that `technique` be the best in that specific `Video game` so we could predict which is the appropiate `method` to use for a given type of `game`.

In `Reinforcement Learning` arguments that are given to the `Algorithms` are as important, if not more important, than the `Algorithm` itself. We can test the best `technique` for a `Video Game` but, if we don't choose the parameters wisely, the outcome could not be an accurate representation of the `Algorithm`'s true potential. This makes another important objective very similar to our last one but, in this case, we'll be focusing on identifying the appropiate parameters for the `techniques` used.

2.2.2. Scope

In this project I will be implementing and testing different `Reinforcement Learning Techniques` however the following tools I will be using during the course of this project, either to implement the

[2]Again, optional software that will make our lives easier, with private repositories.

Algorithms or to test them, are out of the scope of this project and I will **not** implement them.

- **Video Games:** This project's objectives doesn't include the implementation of the Video Games that will be used to test the different Techniques. The reason behind this is that the implementation of the game is not really a factor on how the Algorithm is implemented and therefore not that relevant for this specific project.
- **Neural Networks:** Even though this asset is a huge part on many techniques, the value it gives to the algorithm is not the implementation itself rather than the value of some parameters. Since this project has a due date so short and the reason just given, the implementation of Neural Networks will be out of the scope for this project. Even though this assets will not be implemented by me and I will be using them as a simple user, I will take into account the affects of using an specific implementation to how the Algorithm performs.

2.2.3. Courses Mention

These are some courses that helped me with this project.

- **Artificial intelligence:** This course helped me a lot by giving me the fundamental theorics I needed to start this project. It think it helped me the most the part where Genetic Algorithms were introduced and the assignment I did on a business that used Machine Learning to identify objects in photographs, giving me a base in Machine Learning and making me interested in this topic.
- **Algorithmics:** This one taught me a lot about the importance of efficiency and what tools we have to solve really complicated problems in a reasonable amount of time. It also helped me understand how to properly calculate the complexity of Algorithms so we can differentiate which ones will run faster or save more memory.
- **Compilers:** Lastly, this one helped me realise how much work compilers do so we, the programmers, don't have to worry about minor optimizations and makes us able to create more understandable code and therefore easier to check for errors.

2.2.4. Project and Competences Justification

This project fits well for this specialization because it is based on many of the specialization competences, listed below.

- **CCO1.1[In depth]:** This one is pretty self-explanatory, this competence is the core one for this project since it plays a huge role on the main objective of this project. To determine which `Algorithm` works best for a certain game, I need to be able to properly evaluate its complexity and give the best solution for the problem this project represents.
- **CCO2.1:[Enough]** This competence needed to appear since the objective is to give the best `Algorithm` based on `Reinforcement Learning` and to do that I need to be able to understand the `paradigm` I am working with so I can give the most efficient solution to the problem.
- **CCO2.2:[Enough]** I will need to represent the `Game State` of the game in a way the `Algorithm` would understand and It would be efficient to compute and train an agent. However, I will not go in depth with this competence because my main goal is not focused around how I represent the `state` rather than the efficiency of the `Algorithm`.
- **CCO2.4:[In depth** Once again, this competence is pretty self-explanatory and is really similar to competence CCO1.12.2.4. The core of this project is based on the idea that this competence provides, since its based on automatic data extraction (in this case from a `Video Game`) and treat this information efficiently (in this case, making an Agent learn by itself).

Capítulo 3

Games

3.1. Game 1: FightingGameICE

3.1.1. Introduction

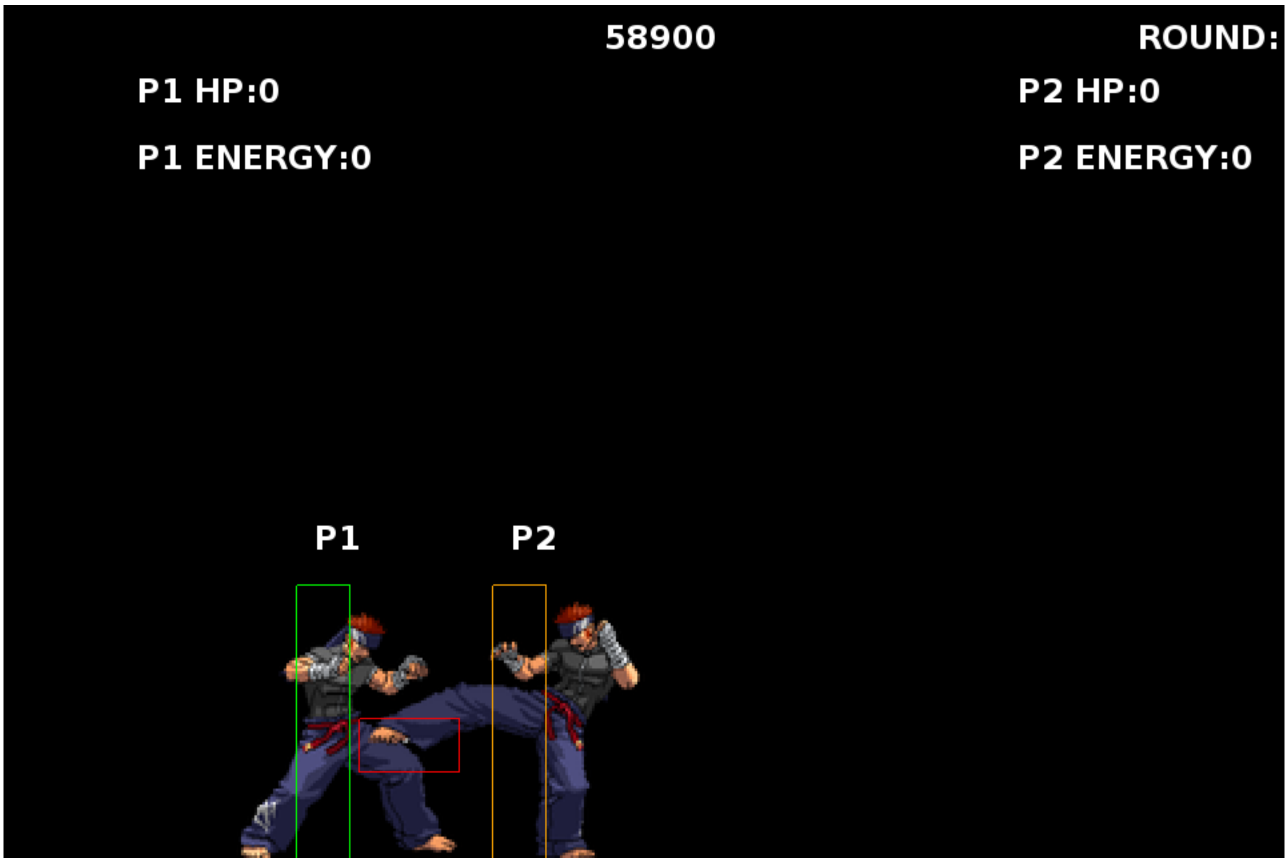
Fighting Game AI Competition is a game developed by Intelligent Computer Entertainment Lab (ICE Lab) from Ritsumeikan University. This game uses game resources from `The Rumble Fish 2` to make it more visuall pleasant with downgraded resolution so it would be faster for AI's to run on this game. The idea behind this game was to give a base game for an AI competition across the world. They have been doing comptetitions with this game since 2013 and doesn't seem to have any plans of stoping any time soon. Each year they have been adding new possibilities to the program to make it more appealing to more people that might be interested in joining the competition. This year's addition has been the possibility of disabling the game window and a `fast mode` which are really helpful when trying to train the agent, which would be even a worse task because of how slow it would be to display the game and train the agent at the same time. Even though this might seem like a great addition, which it is, it was not fully integrated, since the graphics are still loading and so are the sound effects, making it difficult to compute on a remote server such as Amazon.

3.1.2. Game State

Here I present numerous ways to actually represent the `Game state` of this game which were considered to use on this project.

3.1.2.1. Window Pixel Reading

The main idea behind this concept is to simply get a matrix of the colour of the pixels that are displayed on our screen at a certain frame. This can either be done by third-part software or getting the pipeline information that is transmitted into the GPU from the game. Therefore, our agent would be able to see the whole window of the game:



Size Complexity

If we decide to go on and use this kind of approach we will be dealing with a matrix of pixel's colours that has a size of **960x640**, with the default window. We could scale down this window or we could also, instead of getting the whole window, just getting the portion around our player. Nevertheless we would be looking at a window that might need to be at least **256x256** for this to work.

Advantages

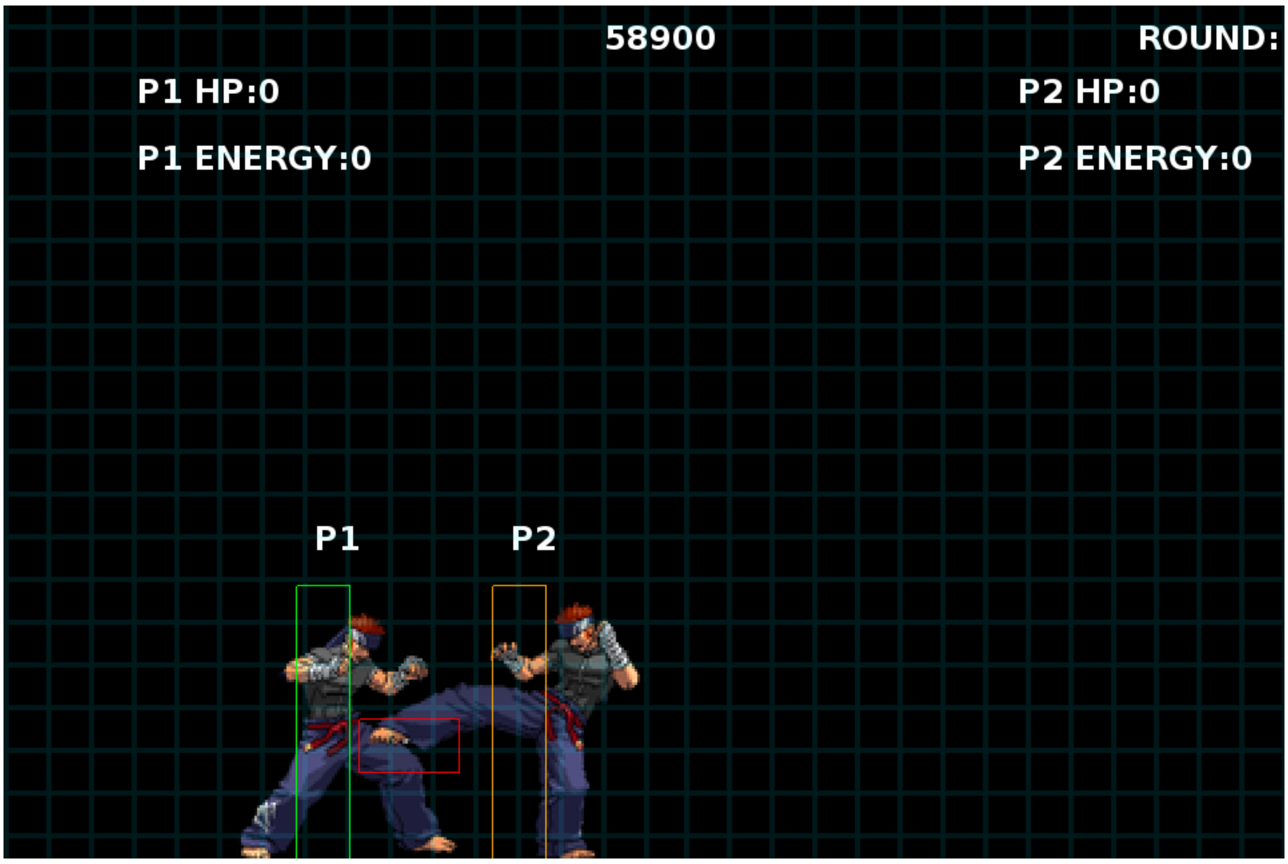
We are giving all the possible information to our agent, having to put little effort on deciding which information is relevant and which is not. Also this strategy emulates really well how humans perceives and learns to play a game, which makes the problem much more appealing, knowing that a program can actually learn very similary to humans.

Disadvantages

The information we are giving to the agent might be `way` too much. The matrix would be really big in any case which becomes a problem when we have to train our agent.

3.1.2.2. Byte Matrix

In this strategy, rather than reading all pixels and passing them to our `Neural Network`, we would divide our window in `tiles`[1]. We can see an accurate representation in the image below.



Size complexity

With this kind of representation we would been feeding our `Neural Network` with a matrix with size **30x20**, where each position in the matrix would represent an object in the actual game. for example: positions filled with 1's would mean that there's an enemy there, positions filled with 2's would mean that there's a projectile in that tile, and so on.

We could apply the same strategy as before, and focus that matrix on what's around our agent. If we take the example before and consider that the **256x256** pixels that surround our agent are the really important ones, we would be looking at a matrix that would take size **8x8** which would be pretty convenient for our `Neural Network` and our `Agent`.

Advantages

Note that this would be a really good strategy for the nature of `Reinforcement Learning`, since we are giving all the window to our agent, and with only that he should be able to learn and accomplish certain level of competence without us having to worry much about its training.

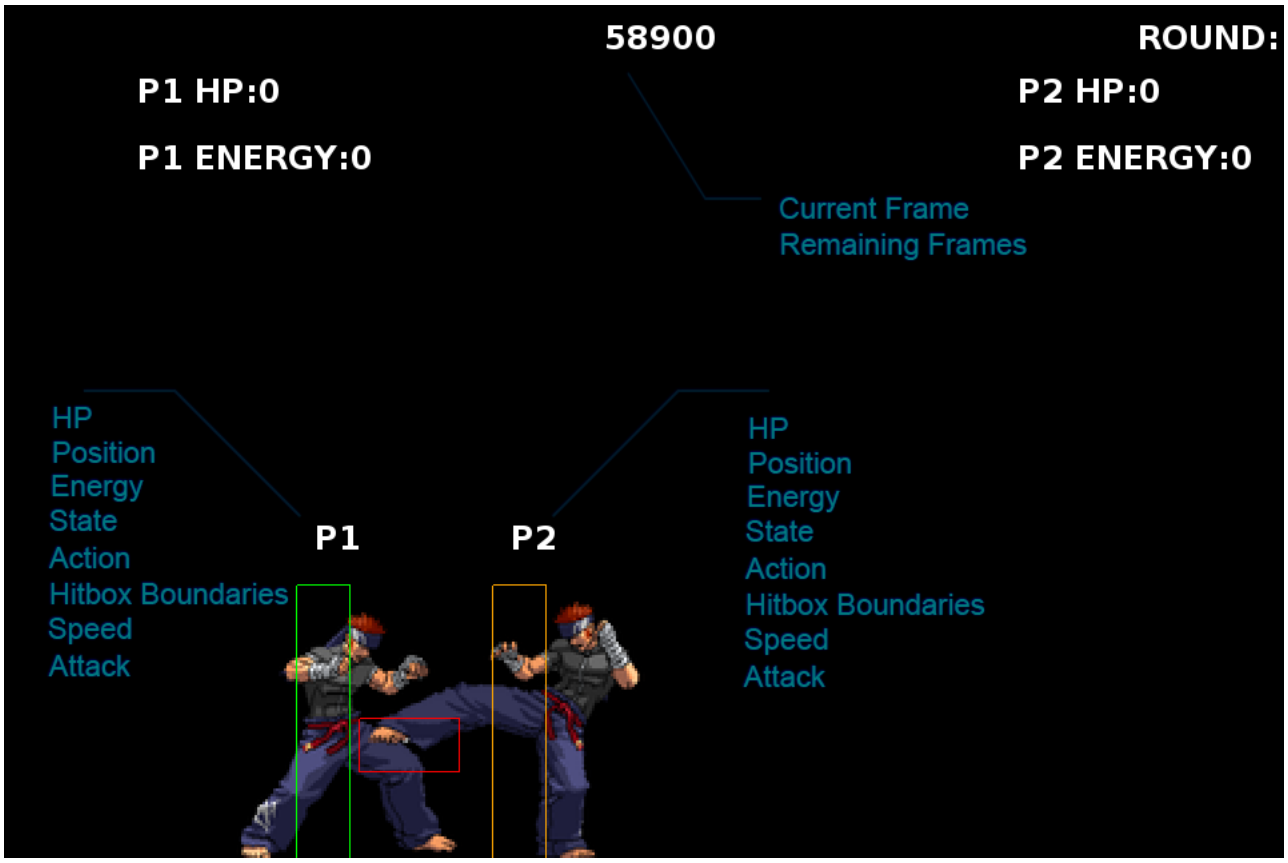
[1] `The size of a tile is determined as the smallest possible unit that the game is capable of measuring.`

Disadvantages

The part where this goes south is that the game doesn't provide this information by itself, and that we should adapt the information we can get from the game to construct this matrix, causing a huge overhead since, for every state, we would have to compute this matrix every single time.

3.1.2.3. All Game Information

In this case we would take absolutely all information that the game provides to us and feed our `Neural Networks` in some ordered fashion. In our case we would use a array which would contain a series of `Integers` representing each a feature from the game state, for example: Position X of the player, Opponent's Hp... We can see a representation on how our agent sees the world in the picture below:



Size Complexity

In this case we would have an array of Integers of WHATEVERELEEMTNS each position representing a feature of the game, as stated earlier.

Advantages

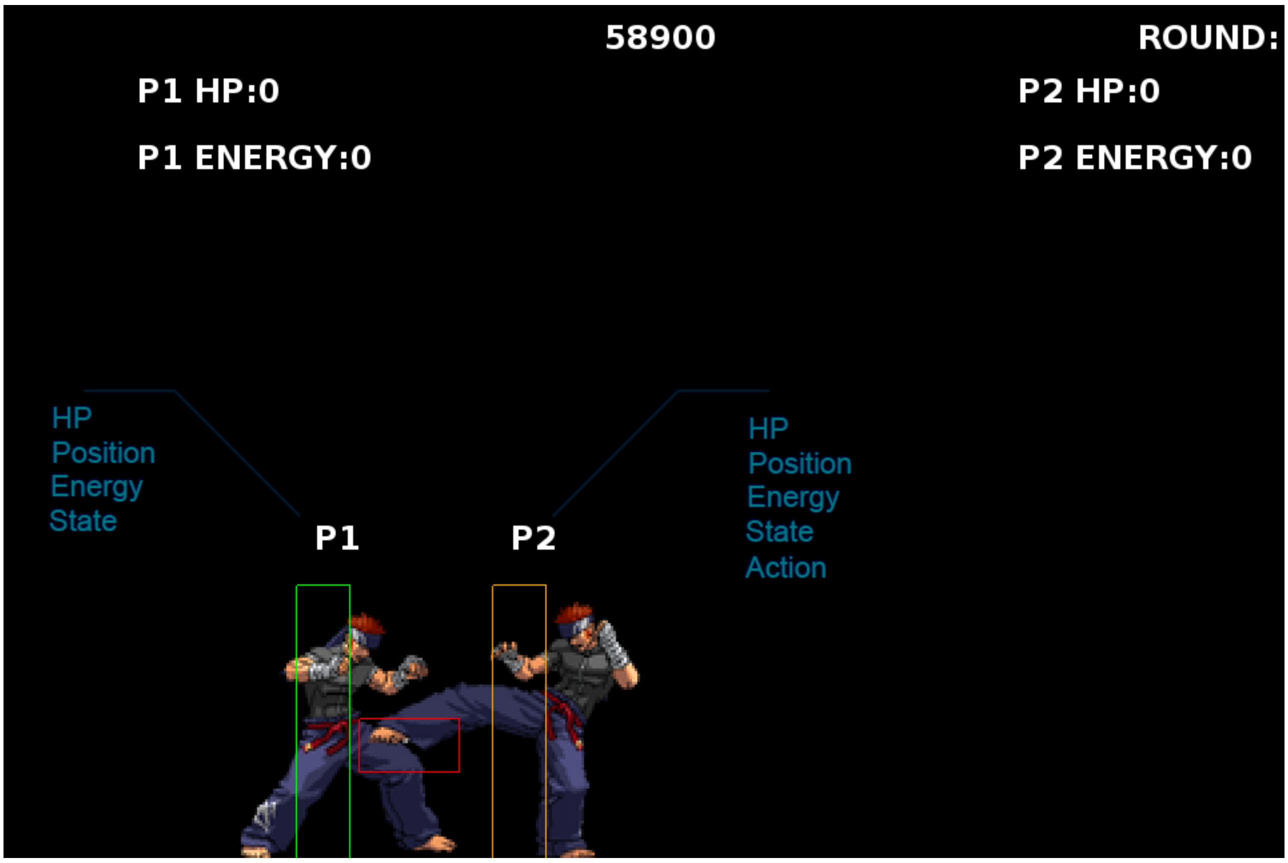
Just as the strategy before, we are using all the information available to us, not worrying to much whether or not a feature is important for our agent or not, opening the door for features that we initially thought weren't relevant to make a much better agent.

Disadvantages

This vector can get out of hand really fast and we would most definitely be giving our agent a lot of information that is not that relevant at all. This might not seem like a problem in theory but it is in practice, since the more information we give to our `Neural Network` and to our `Algorithm`, the more complex the learning process gets, since the bigger the input, the bigger the number of possible states we will have.

3.1.2.4. Relevant Game Information

This one is very similar to the previous one but, rather than taking all the available information, we are taking a subset of it, picking up what I thinks it is more relevant for the agent. This would be the point of view of the agent:



Size Complexity

We would have an `Integer` array consisting of much less positions than the previous strategy, reducing the complexity of the training of our agent, making the training simpler with better results on the short term.

Advantages

Since we have such reduced number of inputs, the learning process of our agent will be considerably faster, since it won't have that much complexity, focusing only on things that really matter for the decision-making.

Disadvantages

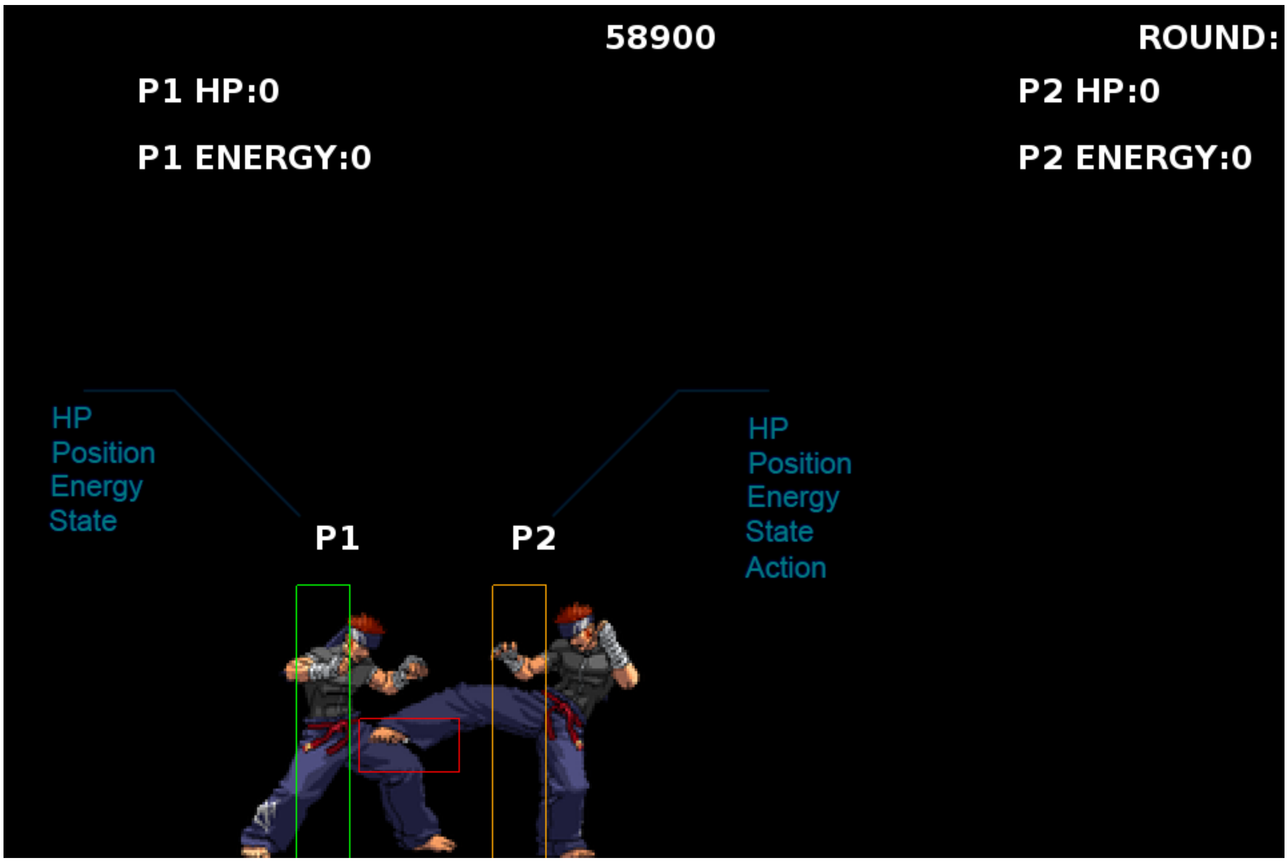
The agent will not see the whole picture forcing me to pay close attention to what really matters in the game and what doesn't, making the process a bit more tedious and not so independent. Another risk is that we might reduce a bit too much the game state, losing valuable information for the agent.

3.1.2.5. Final Decision

The strategy we will be following for this game is **Relevant Game Information**, the reason being that it gives all we need for this project. The reduced number of inputs will help us to deal with the computing issues we currently have, making us work a bit more on deciding which are the game features that will matter the most for our agent.

We are not considering the options of reading each pixel of the window, or a portion of the window for what's worth, because not only would result in really big matrices but, the procedure to get the pixel's colour is also quite time consuming for our algorithm, resulting in an important loss of performance both at `training` time and `executing` time

This is what our agent will see at a certain time of the game:



To get more specific, here's a list of the features included in the agent:

- Player HP
- Player Position X and Y
- Player Energy
- Player State
- Enemy HP
- Enemy Position X and Y
- Enemy Energy
- Enemy State
- Enemy Action

This leaves us with an input consisting of an array of **11 features** which should be more than enough for our agent to properly learn the basics of the game.

3.1.3. Actions

In `FightingGameICE` there's a lot of actions that once can perform using `ZEN`, the character that our agent will use for its training.

These actions are performed via the class `CommandCenter` which is provided by the developers of the game. This class is really helpful because all that is required from the programmer is to select an action from the **58** possible actions, and the `CommandCenter` will do the rest, performing the action in-game and giving you feedback about when you are able to make an action again.

Once again we have a dilemma here regarding the size. The more actions our agent is able to make, the more output size our `Neural Network` will have and the more complex to train will get. Therefore we need to cut down some of those actions to a relatively low amount in order to realistically be able to train our agent. These actions will be stored in an `enum`, called ReducedActions, and will be used in substitution of the `enum` that the game provides.

The selected actions for our agent to perform are **11**, which are:

- BACK_STEP
- STAND_GUARD
- STAND_A
- STAND_FA
- STAND_B
- STAND_FB
- STAND_D_DF_FC
- CROUCH_GUARD
- CROUCH_B
- CROUCH_FB
- THROW_A

3.1.4. Rewards

This section is probably the most important one, since it will affect the most to how our agent will learn with the base that we've provided him with. Luckily, for this game the objectives are pretty clear: Hit the opponent and do not get hit.

That's why we don't need to invest a lot of time in a complex reward/punish system for our agent. Since the nature of this project is to keep things as naive as possible, we won't be trying to make a really complex reward system. This is basically how it's going to work:

- **+damageDealt** Everytime we hit our opponent.
- **-damageReceived** Everytime we get hit.

This simple system should make our agent understand that, while you are making actions that hurts your opponent, you are doing great in the scale on how hard you are hitting him. On the other hand, if you are getting beaten up by the actions you are performing, you will be getting negative rewards for those actions. This understanding of the game should be enough to implicitly make him win the game, since the more he's hitting, the more reward he will get.

For the sake of this project, the system does not take into account either you win the game or you lose the game, or if you are not hitting your enemy at all. The first feature, would make our agent learn faster to win games but it might backfire, and make our agent just run away and blocking his opponent while he's wining, which wouldn't be that much interesting to watch. For the second feature, we would be rewarding our agent for constantly trying to hit the opponent, which is not a good objective to have in a fighting game, and it could also create a lot of confusion on why a past action once struck very hard and one right now doesn't do anything at all in an apparent similar state.

For those reasons I think that the strategy I picked was the more balanced and should provide a pretty decent show, and hopefully win the game.

3.1.5. Learning Strategy

Training methodology

This game was actually pretty hard in terms on how to train it. The first idea was to use the parameter `--disable-window`, provided by the developers to train the agent in a server from `Amazon AWS` which was free for a limited period of time. However, this turned out to be not so viable as I inicially thought for the fact that, even though the window was disabled, it was actually loading up all the textures and sounds of the game, making it nearly impossible to train on a server unless you create a `dummy soundcard`. After spending lots of hours trying to figure out how to do that, and delaying the whole project, I decided that a better idea would be to just train it on my laptop, with a limited range of values for the parameter `gamma`.

To train it on my laptop I created an script which take as parameters, the number of games to be played and the current gamma that's being used. This script, will save the `neural networks` at each iteration and saving them with the gamma parameter as a suffix, this way I will be able to store all the `neural networks` for all the values of gamma in the range. Also, while I am executing this script, I will have another one that will monitor the game to check if it becames stuck for any odd reason, making the training process much more automatic and parallel. The game will be executed with the parameters `--disable-window` and `--fastmode` to improve the performance of the training sessions. Also the agent will be matched against different AI's which were took from the competition of last year.

Gamma Value Range

As I have commented previously, I will need to have a subset of the possible range of `gamma` values. The range for these values will be **0.1-0.5**, the reason being that, for this game and for the strategy we're going for, it is more valuable the close to inmediate reward rather than the long-time reward. For example, we're interested on giving credit for the actions that gave us reward in the following **2-5** states, not on the ones that gave us in **20** because those are not necessary linked to the action from **20** states ago.

3.1.6. Agent Implementation

After doing all the necessary set-up to actually be able to compile and run the game and the agent, it comes the time to start adapting the `Q-learning` and `SARSA` algorithm, to the kind of agent we are working with. To make it as general as possible so I could reuse the majority of the generated code for also the other games, we have some important classes that help us make the algorithm inputs and outputs as general as possible, making the process of creating a new agent a kid's game.

These classes, or structures, are **FeatureVector**: which will be the variables we'll be feeding our `Neural network` with and **ReducedActions** which are the actions that are available to our agent and thus, the actions that the agent will be able to perform and keep track of their importance in the game.

Let's get a better look at what classes and structures makes the agent be able to use the algorithm.

3.1.6.1. FeatureVector

As stated earlier, this class is the responsible to order and process the game data and store it in a vector so it can later be fed to our `Neural Network`.

This class is really simple, yet so important, it basically has a method called **convertState** that is also called when an instance of this class is first created. This method receives as a parameter the current state the game is processing, and return an `array` composed by `doubles` that represents that same exact state in a way where the `Neural Network` will recognize it and be able to train it efficiently.

3.1.6.2. ReducedActions

This is an `enum` where basically maps an action with an integer number. This makes the retrieval of information from the `Neural Network` much much more direct, since we'll just be accessing a vector where each position is a possible action that we might take at a given time.

3.1.6.3. CommandCenter

This is a powerful class provided by the game which let us command and action to our agent via a simple string, which is actually the name in the `enum`, and it performs all the hard work for us. This class will make our agent do whatever action we told him to and, once is finished and it is ready to process another action, it will notify us via a `boolean` variable.

3.1.6.4. NNAlgorithms

This same structure and functionality is defined for both the algorithms we are discussing in this project. Their respectives name are `NNQLearning` and `SARSA` for the one holding the `Deep Qlearning` and `Deep SARSA` algorithms, respectively. These classes are the ones that are responsible for the algorithm logic to work and are the ones that heavily rely on the usage of the `Neural Networks`. This class is the responsible of configuring the `Neural Networks`, parameter values, training the `batches` and how to train those batches. Also this class is heavily used from `AgentDeep` class, having a really close relation.

This class is also the responsible for the logic of the ϵ`-greedy` policy, keeping track of which epsilon to use, and returning an action accordingly. I did also included here methods to save and load the `Neural Networks` in an easy way.

3.1.6.5. AgentDeep

This is what would be the `main` class of our agent. This class is used by the game to use the agents on a general label. This class needs to extend from `AIInterface` class, which is provided by the developers.

Our class will have to implement the following methods to fit the requirements of an agent described by the competition organization.

- **initialize**: This method will be called at the start of each Game. This is the place to make he first initializations such as, initialize our agent, load our saved `Neural Networks` or initializing the player information.
- **getInformation**: This method is called at each frame of the game, and its meant to feed information to our agent such as: if the game has started, information about the game state (player's position, hp...).
- **processing**: This method is called just after the previous one. This is the one in which we will be all our computing and training, since we'll have all information about the game state available and all that is left to do is decide an action.
- **input**: This method returns the `Key` that needs to be pressed.
- **close**: This one is called just after finishing or closing the game. It's purpose is to save the `Neural Networks`.
- **getCharacter**: Which states the player our agent will be playing for the game.

getAction Method

In `processing` method is where the most important part of the algorithms is located. The workflow that follows is actually the algorithm.

First, we calculate the reward from the previous and current `Game State`. We need those two because if we recall the reward system we have for this game, the rewards are the difference in between the hp of the player/enemy in two consecutive states.

Once we know which immediate reward we are getting for being in the state we are in, we proceed to select an action using an ϵ`-greedy` policy, which logic is actually located in our `Agent` class.

Then we create a new `Transition` using the information about the previous state, the current state, the reward we are getting from being in the state we are in and the action we decided to take. This is later given to our agent so he can use that information to train our `Neural Networks`.

Finally, all that's left is to set the `CommandCenter` class to perform the action we decided it will best fit the situation using the method `commandCall`.

3.1.7. Game Metrics

To evaluate the performance of the algorithms in this game, we'll be taking some information of the execution of the agent so we can have data on how well it is performing. The data will be the median value of 30 rounds of the following metrics:

- number of won rounds
- number of hits dealt to the enemy
- number of hits taken by the enemy
- number of damage dealt to the enemy

- number of damage taken by the enemy

With this selection of data, we should be able to tell if our strategy of only accounting for hits is viable and whether or not the algorithms perform well with the parameters that are being used. It also allow us to make much more objective conclusion on the performance of the game, focusing on the results rather than on subjective action making of our agent.

3.2. Game 2: Mario

3.2.1. Introduction

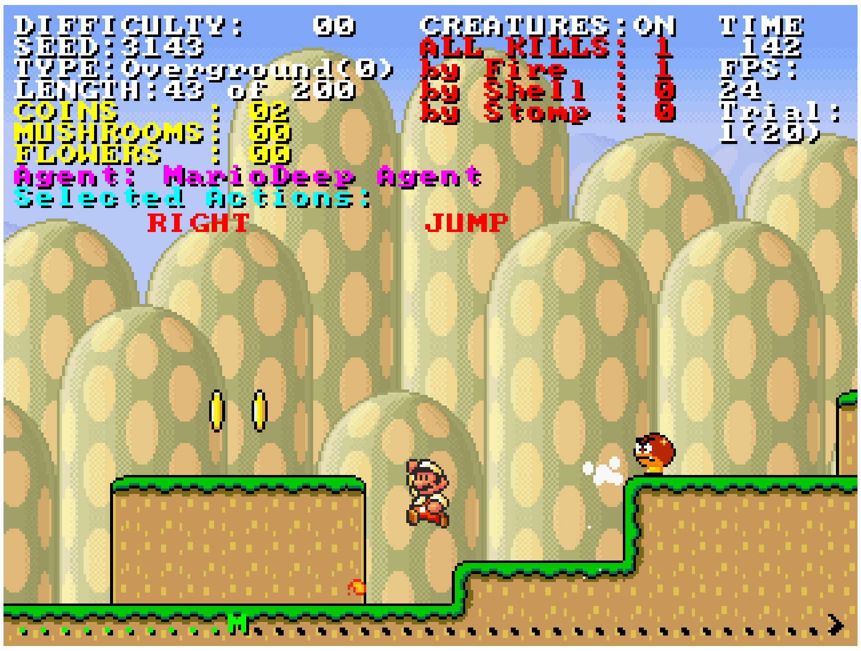
Just like in the previous game, this one was also developed in the hopes of providing a good base to make AIs competitions. Sadly, the competitions are no longer being held by them since 2012. The current official page for the competition is not well-updated and has some broken links that lead nowhere, making it impossible to download the source code from there to start programming the agent. Luckily, they still have the old version of the web, which is from 2009 and has all links updated and everything works like a charm.

3.2.2. Game State

The options for this game are very similar to the other one, but the complexities and best strategies change dramatically due to how the game is built.

3.2.2.1. Window Pixel Reading

The procedure would be exactly the same as in the `FightingGameICE` since the developers didn't leave any way to get the image directly, we would have to use the same methods that I considered above.



Size Complexity

The window on this game is really small, which is actually on purpose so people would be able to train the algorithm with this strategy in an easy fashion. Another reason why they did it so small is so the training and execution time of IAs are greatly reduced since the textures are all low-detail and small. The same reasoning from the previous section applies to the size complexity of this game. We've a full window of **320x240** pixels, and another one that could be reduced to around **162x124**, considering that we take some decent height from mario and just focus on what he has in front of him.

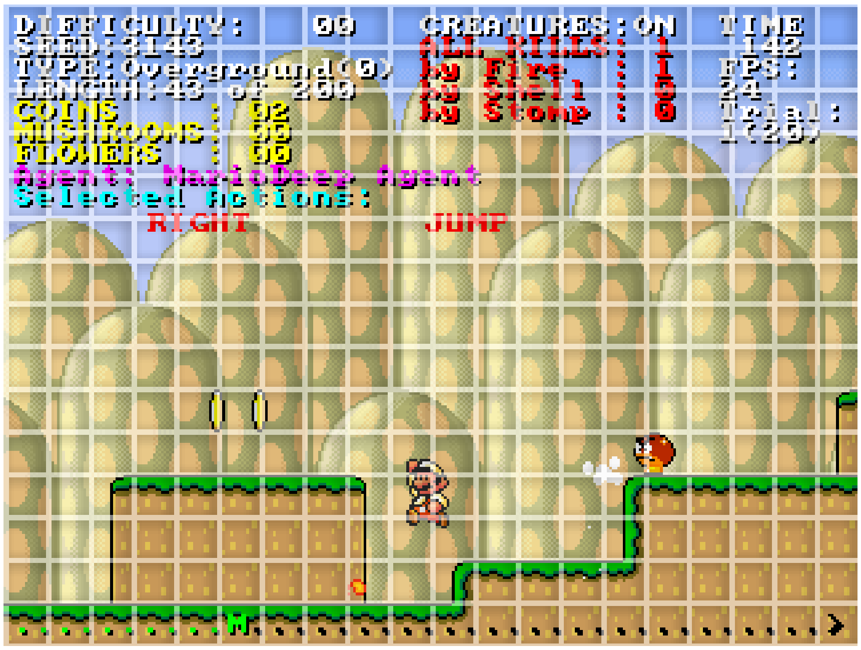
Advantages

If we've already applied this strategy to our past game then it would be relatively easy to make our third-party software get our new game's window rather than the old one and feeding it into our algorithm. Another advantages are already commented on the previous section, but I would highlight the fact of how little effort this leaves us on deciding which information is relevant and which is not.

Disadvantages

Even though we are talking about a much smaller size complexity than in `FigthingGameICE`, it is still too much to handle four our current computing power, becoming way too many input parameters for our `neural network`.

3.2.2.2. Byte Matrix



```
DIFFICULTY:    00      CREATURES:ON     TIME
SEED:3143              ALL KILLS :  1     142
TYPE:Overground(0)     by Fire   :  1   FPS:
LENGTH:43 of 200       by Shell  :  0   24
COINS      :  02       by Stomp  :  0   Trial:
MUSHROOMS  :  00                        1(20)
FLOWERS    :  00
Agent: MarioDeep Agent
Selected Actions:
      RIGHT             JUMP
```

Size Complexity

This time around we will have a much reduced matrix to represent the current state of the game. We'll have a matrix of bytes with size **21x21** that can be lowered to **15x11** if reduce the line of sight of our agent.

Advantages

The advantages are pretty much the same as in the previous game. We will get pretty much the benefits of reading the full window with a much more reduced input matrix, making our algorithm faster and easier to train. Also, in this game the developers actually give us a byte matrix, so we don't have the problem that we had before, were we had to create that matrix.

Disadvantages

If we decide to take the byte matrix of the whole scene, it might still be too big, and we would need to somehow reduce the field of view of our agent in order to have a much more efficient matrix and implicitly, a much more faster training. If we decide to take a reduced byte matrix, we should be careful and not make it too small, in which case we would be missing important information about the game.

3.2.2.3. Game Information Based

The same principles as the previous game applies for this strategy but, here it is not so clear how the actually game could be represented with such strategy, giving a lot of work to the programmer to see which is the best approach to it. This strategy actually contains the both strategies that depend on game information from the previous game, due to the highly number of possibilities depending on how we represent the world.

Size Complexity

The size of complexity of this strategy is actually not trivial at all, it will highly depend on how we are going to represent this game and how the level is actually composed of. We could have information such as: Mario position, where does this platform end, where does the next one begin, for each possible enemy on the level where it's located...

Advantages

We have much more precise information about the level and about `Mario` in general, giving to our agent accurate information will result in much more accurate movements and decisions, if trained for long enough.

Disadvantages

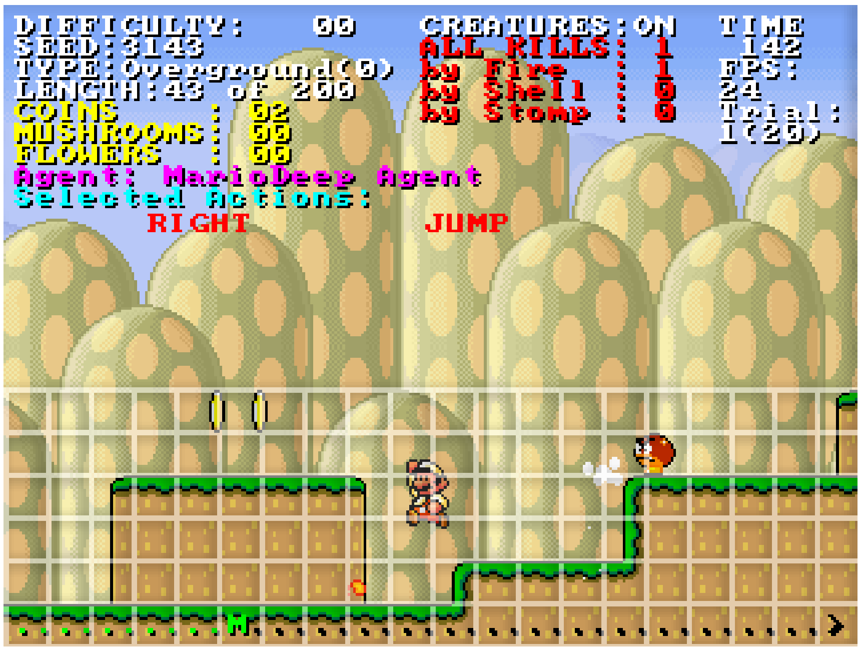
We have to really think how the world should be represented in a single vector of a constant size, for the `neural networks` to be useful. Also in this game we have exactly the opposite situation as in the previous one, if we decide to choose this option, we will have to transform what the byte matrix is providing to get information about level and such, causing a overhead when we were to train our agent.

3.2.2.4. Final Decision

The option I will be going for this time is for the **Byte Matrix**, since it is the one that gives us the most benefits `by far`. With this strategy, we don't need to worry about how we are going to represent the world in a single fixed matrix as we should do in the `Game Information Based` method since it will be already be provided by the game.

To further improve the `training` efficiency of our agent, I will be reducing the field of view of the agent. This way the `Neural Networks` will be fed with a much smaller matrix, being able to train faster and to reduce the number of pairs `state-action` which will greatly improve the learning rate of our agent, by giving him a simpler understanding of the world. This mentioned matrix will have size **15x11** which should be more than enough for our agent to get a pretty decent view of the world.

This is a representation of what our agent would see at a given time:



3.2.3. Actions

The system to actually input actions to our agent in this game, is based on a vector of `boolean` which purpose is to have information about which buttons are pressed. I had to adapt that and make some sort of logic to actually transform actions into a vector that the game can process. This way we are actually mapping each action, that the algorithm understands, into some series of key presses, that is what the game understands.

To do this I simply created the method `performAction` which is located in the `ActionMapper` class and receives one of the `enum` that represents the action to take. Essentially this is just a `switch,case` that, given the action you want the agent to perform, it returns the vector of `booleans` that makes that action performable.

For the actions that are actually available to the agent I followed a similar idea from the creators of `FightingGameICE`. I created an `enum` so I can have an easy way to refer to certain action and an easy way to index that action. This `enum` has **10** fields, which are:

- RIGHT
- LEFT
- DOWN
- JUMP
- SPEED
- RIGHT_JUMP
- LEFT_JUMP
- DOWN_JUMP
- RIGHT_SPEED

- LEFT_SPEED

3.2.4. Rewards

The reward system for this game is a bit more complicated than the one applied on the previous game. The reason being that there's much more to consider giving the nature of the game. On the previous one, we didn't care too much about rewards that were too far away on time but in this game, it is actually quite the opposite way. For this game we will be considering the following reward/punish system for our agent:

- **+4.5** while you are moving towards the end of the level.
- **+6.0** while you are moving `FAST` towards the end of the level.
- **-2.0** while you are moving away from the end of the level.
- **-2.5** while you are moving `FAST` away from the end of the level.
- **-2.0** if you are not making any progress.
- **-300** either you die during the level or you run out of time.
- **+200** if you manage to complete the level.

With this reward system we will make our agent learn the simplistic logic about the game: **move to the left**. For the nature of the algorithm, we need to actually punish our agent for not moving, the reason is that if you don't punish that behaviour he will choose not to move at all, since it doesn't punish him, even though he knows there might be some states that might give him a better result. This behaviour will change at the end, since we are punish him anyways for not completing the level on time but that would take a long time, and since we are already having computational time issues, this will make our agent learn significantly faster.

I've also added a higher reward / punish if you move faster than normal (that is, pressing the SPRINT in order for the agent to be able to learn that the faster the travels to the right of the screen, the more reward he will get. This should implicitly mean that, the faster you finish the level, the more reward you receive.

3.2.5. Learning Strategy

Training methodology

Since I am trying to compare the two games among them, I will be performing similar training strategies for both games, even though the developers for this game actually took their time to implement versions of the game that are able to train efficiently on servers.

Either way, this game has way less issues than `FightingGameICE`, making it easier to train even though I will not be running the training sessions on a remote server. The training sessions will consist in iterations of executions of the game. At each iteration, the game will be executed with a limited time of **3** minutes to complete the game. After each iteration, the `Neural Networks` will be saved and stored accordingly to their gamma value, just as in the previous game.

Gamma Value Range

This time, the actions that we took in the past, are heavily correlated on the position we are currently in. This is why for the range value for this game, the values will be higher than the previous: **0.5-0.9**. The theory behind this is that, if we finish the level, we should give credit to `all` actions that has made us win such a juicy amount of reward, and the way to achieve that is to use high `gamma` values. Also keep in mind that this is a game where the most reward and punish, are received at the end of the iteration so we'll need a high `gamma` value to make sure this reward propagates fast through all our states.

3.2.6. Agent Implementation

3.2.6.1. FeatureVector (Byte Matrix)

In this case we won't be having a class to process the game state and make the mapping so the `Neural Network`, at least not explicitly. The reason behind that we actually already have a representation of the game state in form of a matrix so the `Neural Network` will be able to understand which is the current game state without any extra processing, being able to produce values that we are going to use later for the training.

This byte matrix will be given by the game and reduced so it has less elements on it, making the learning process much easier and simpler, since we will be having less possible states. This will be explcained later in the document, in the `MarioDeep` section.

3.2.6.2. Reduced Actions

For this game, I had to create the `enum` containing the different possible actions to perform, since the game only undestands button presses. This is why I had to implement some kind of logic to be able to abstract those button presses into actions, just like it has been implemented in `fightingGameICE`, this way the process of selecting a new action and processing the information will be greatly simplified.

To do this I created the class **ActionMapper** which contains a method called that receives a `ReducedAction` name and return a vector of booleans representing the buttons that the agent needs to press to actually perform that action. To make this as efficient as possible I just implemented it as a simple `switch;case`.

3.2.6.3. NNAlgorithms

Since I've had already done the algorithms class pretty general enough, I didn't need to change many things in this class. The only important part that I had to change was the configuration for the `Neural Networks` so they would handle the new input size, since it will no longer be generated by a `Feature Vector` class. I also needed to make sure that It was using Mario's `ReducedActions` and not the other one to avoid errors relating the selection of actions. Regarding the `transitions` I also had to change them so they would adapt to the new type of `FeatureVector`.

3.2.6.4. MarioDeep

This is the main class of our agent. This class will be called by the game to ask for actions and interact with the environment. This class will have to implement the following methods in order to be a valid agent.

- **reset** This method will be called at the begining of each executing of our agent, here's the excellent location to initialize all the necessary parts for the agent in order for it to work, from loading the `Neural Networks` to set certain intern values of the algorithm.
- **getAction** This is the most important method. It will be called at each iteration, allowing our agent to make an action. This method will be explained in depth below.
- **getType** With this method we set the type of our agent. This is an internal feature of the game, so it can know whether is an agent or a human.
- **getName** and **setName** Thanks to these two methods we can get and set the display name of our agent while its running.

getAction Method

The work that `getAction` does is very similar to `AgentDeep's processing` since it's where most processing and data is going to be entered to our algorithm. The objective of this method is to provide the algorithm with enough data so it can train and give the agent an action to perform based on all the data that has been already processed.

First, calculates the reward we get from being in the state we are in. To do that he gets the current Mario's position and compares it to the last. Then he uses the `reward system` that it's been decided for this game, and returns a value on how much should the algorithm reward or punish the agent for the state we are in.

After getting the reward, just as we did on the previous game, we get an action, using an ϵ`-greedy` policy. Then we make the agent learn the transition and lastly, we give our `ActionMapper` the action we want to perform while setting the variables `previousState` and `lastMarioX` with new values for the next iteration.

3.2.6.5. MainRun

Since in this game we actually have the source code we can configure our training session inside the current program making it much more efficient. This is the main class from where our agent will be called and where all the environment variables of the game will be set.

Apart from the default configuration and actions, I've added some logic to be able to repeat the same level many times. We are also able to make adjustments for our agent in this part, having a much more controlled environment where we can adjust things in a pretty much bigger scale, opening a world of possibilities.

Regarding the environment configuration, at first it would just randomly execute in between 3-4 levels of different difficulty. I've changed that so it would only be able to generate one level with a fixed seed and a fixed difficulty, this way the agent will have a much more stable environment helping a lot on the training.

This class is a great place to make some adjustments from outside the training session to our agent. Things like giving him a positive reward if he has won the game becomes a really easy task, compared to having to know in-game if the agent won or not. To accomplish this I simply overwrite the last transition with the corresponding reward for winning/losing the game. This is also a great place to load and save the `Neural Networks` much more efficiently.

3.2.7. Game Metrics

These are the game data that will be extracted and averaged over 20 executions of the same level:

- number of successful executions (wins)
- number of kills from the agent
- Time left at finishing level
- in-game score which is basically a sum of the last metrics.

With these metrics we will be able to have proof which is the algorithm and parameters that made our agent finish the game the fastest, whether or not its consistency on finishing those levels was acceptable and whether or not if he felt it was important to kill the enemies to avoid possible future threats.

Capítulo 4

Results

4.1. FightingGameICE

Here are some of the results that I found interesting from the execution of 30 rounds of the game with our trained agent against a completely random opponent.

4.1.1. Parameter Testing

In this section I will be sharing some of the results among the different possible parameters for each algorithm, lets begin with SARSA first.

4.1.1.0.1 SARSA

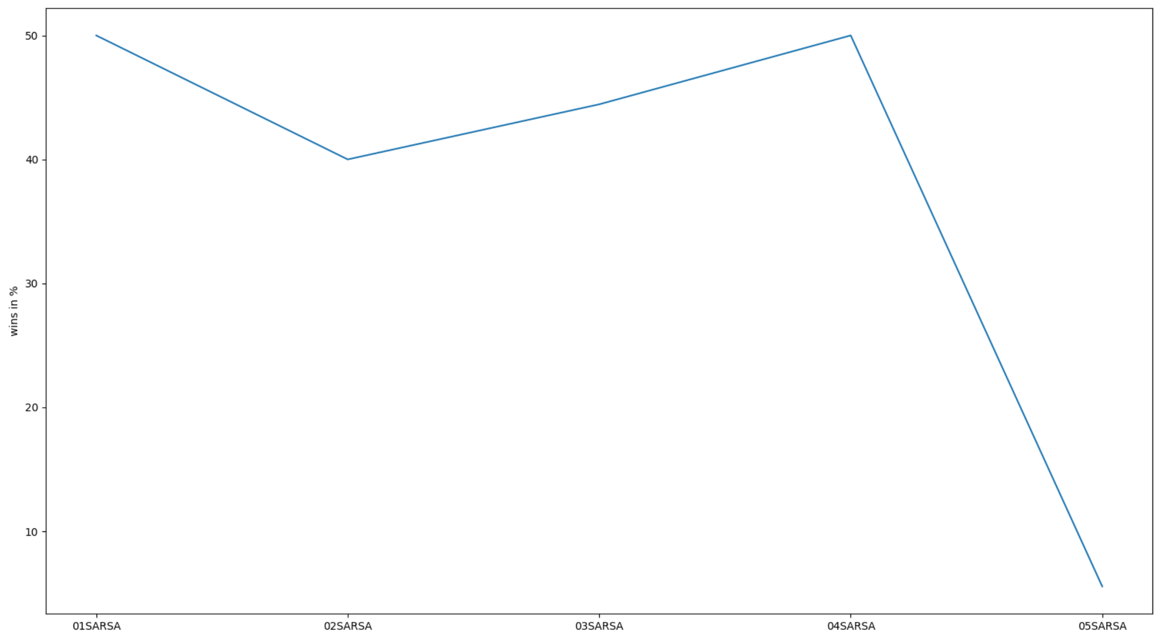


Figura 4.1: Percentatge of wins for SARSA

In this graph, we can see the percentage of wins that the agent had during all the executions. As we can see, it is **not** good at all, being under the 50 % winrate on all parameters values. Also note how `SARSA` with parameter `gamma` 0.5 has even under a 10 % winrate. This graph then is basically an indication that, if we use `SARSA` algorithm for our agent, we won't be getting much better than just using a random action-driven agent.

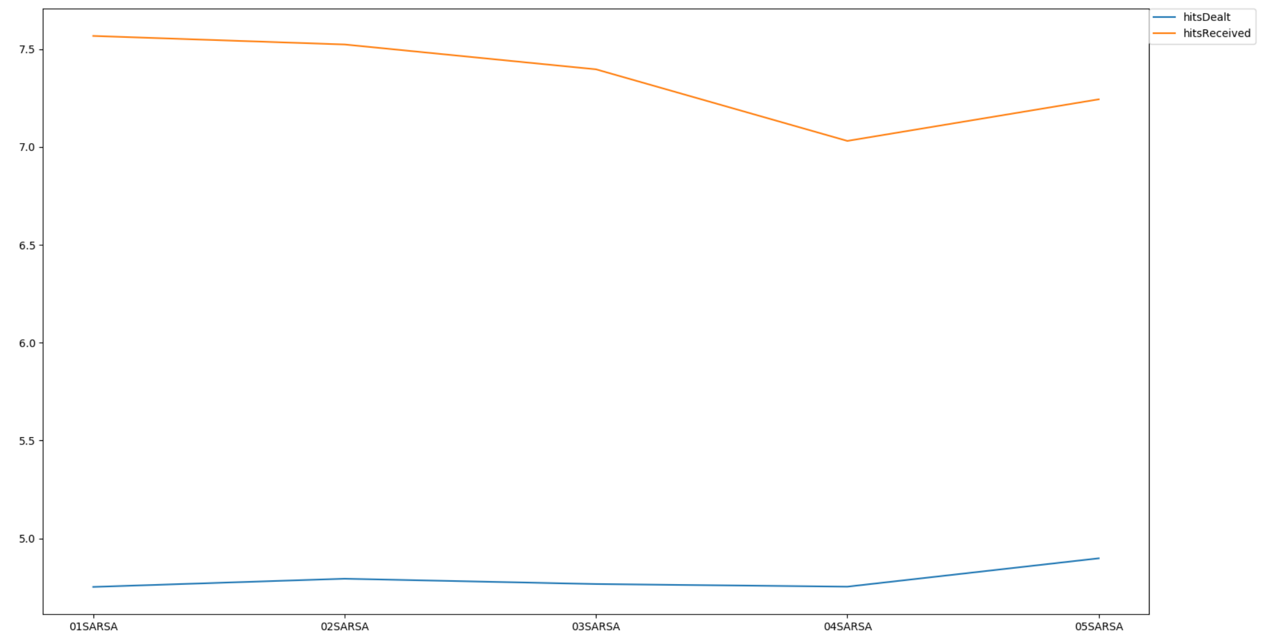


Figura 4.2: hitsDealt vs hitsReceived for SARSA

As we can see, here's a good explanation on why our agent is so bad in all of those executions. The agent hits way less than the opponent, resulting in a drastic difference between the potential damage the opponent can deal and the potential our agent can. Surprisingly, on `SARSA 0.5` the agent hits more than the average but it is the one that loses the most, meaning that it possibly plays way too aggressive and receives a lot of damage from strong hits of the opponent.

4.1.1.0.2 QLearning

Similarly to the previous section, lets go through the same process and see how the parameters affected our agent for this algorithm.

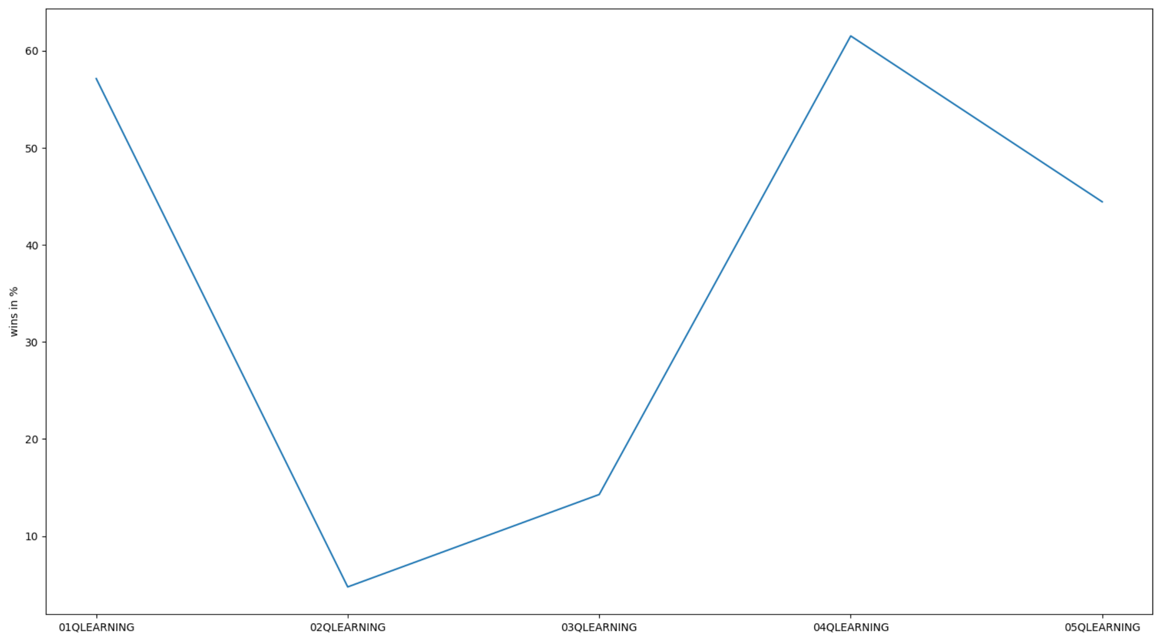


Figura 4.3: Percentage of wins for Q-Learning

This one looks like has way better performance than the other algorithm. As we can see we've instances that are very close to the 60 % winrate, even surpassing it by a bit when using `QLearning0.4`. Sadly, not all instances were great. We can see how `QLearning0.2` and Qlearning0.3 struggle quite a lot against our random opponent.

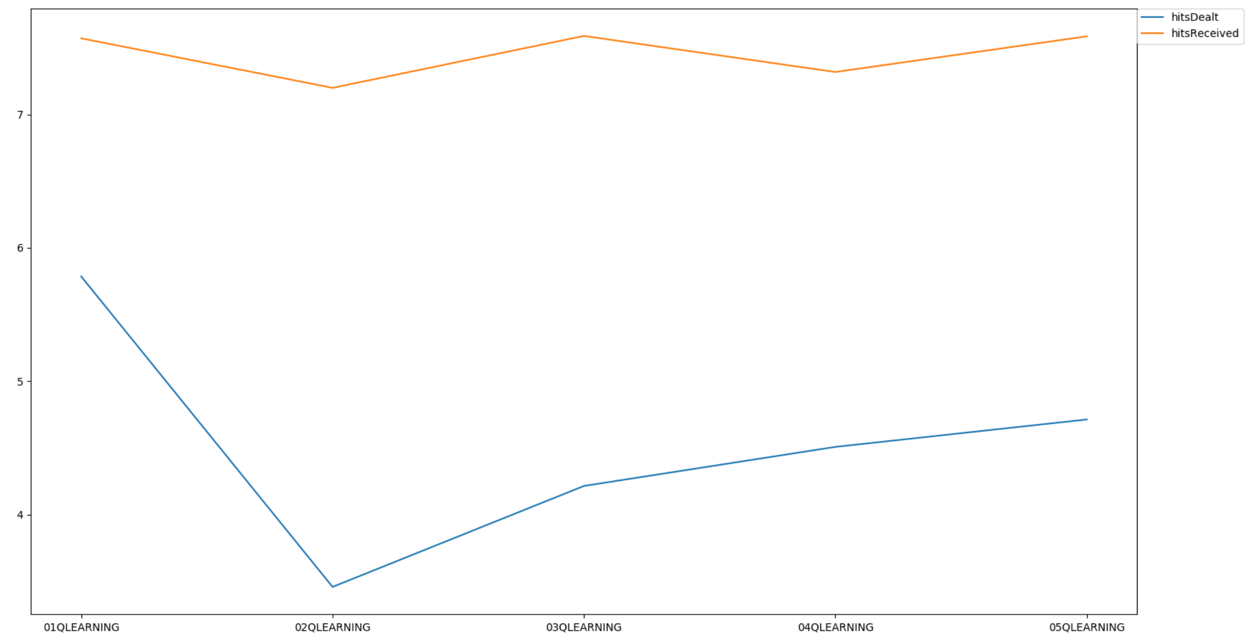


Figura 4.4: hitsDealt vs hitsReceived for QLearning

Amazingly, we have a very similar situation where we are hitting more or less the same (Except on `QLearning0.1` which we actually hit quite a lot compared to other instances). This can be explained if our agent is actually waiting to strike hard, rather than strike a lot of times dealing little amount of damage each time. Also since we are not penalizing our agent for not hitting the opponent, it is possible that the agent tries to avoid contact unless he thinks he can get a good hit.

4.1.2. Algorithm Testing

Now it's time to bring together both of the graphics and compare how well the algorithms did between each other, lets start with the wins:

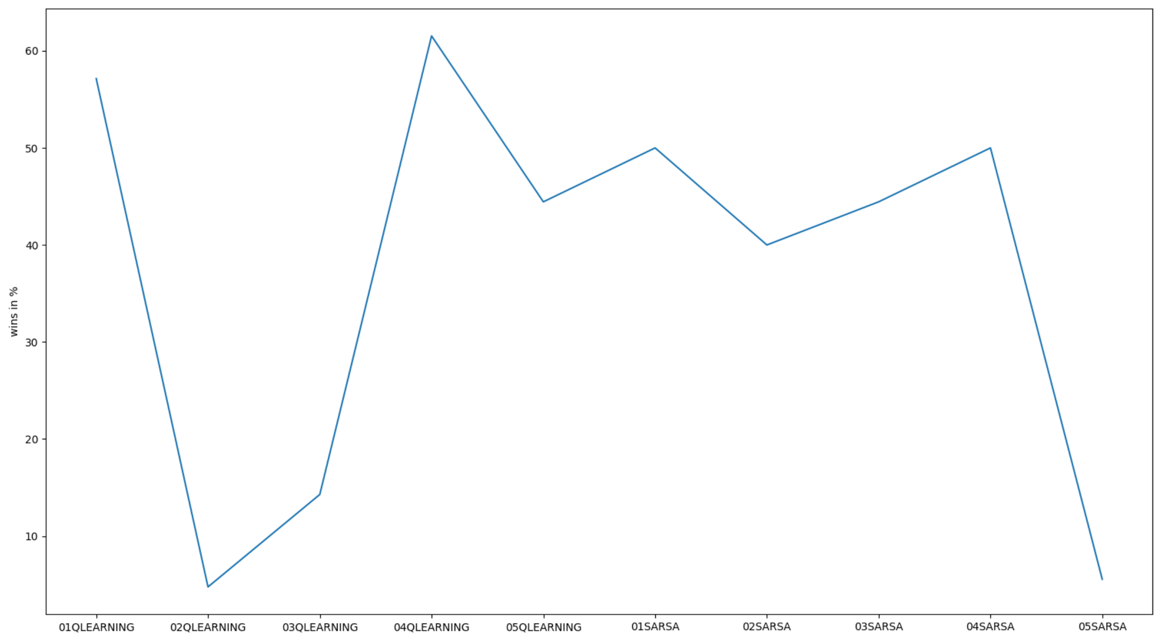


Figura 4.5: Percentatge of wins for both algorithms

As we can see, `SARSA` instances perform better in average than the ones using `Q-Learning` as their algorithm. However, we can see how some `Q-Learning` instances take a huge lead from SARSA, going up to almost 60 % win rate. Having `Q-Learning04` as the agent that got most wins in all the instances that were tested, followed very close to `Q-Learning01`, possibly meaning that the value functions affect to the performance either immediately or a few steps after the action was made (which might happen with projectiles)

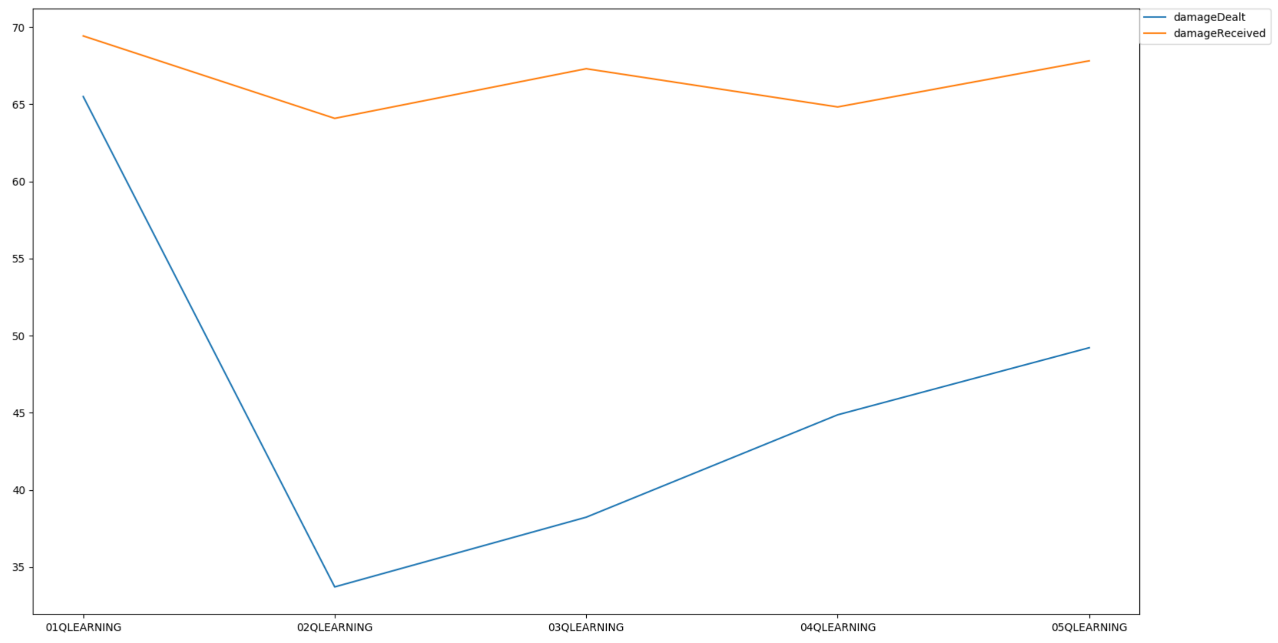


Figura 4.6: hitsDealt vs hitsReceived for QLearning

To mix up things, we are now looking to the average damage our agent did per round, rather than the average hits per round. As we can see, the average is very very similar among instances, excluding `Q-Learning0.1,2 and 3`. One that really caught up my attention is `Q-Learning0.1` which performs near 65 damage per each round, very close to our opponent.

4.2. Mario

The results that will be shown in this section are the average from 40 executions, using the same seed and time as in the training to complete the level.

The first idea for this section was to also give the results in the range that I decided would work the best for this game. However when I plotted the results I found this:

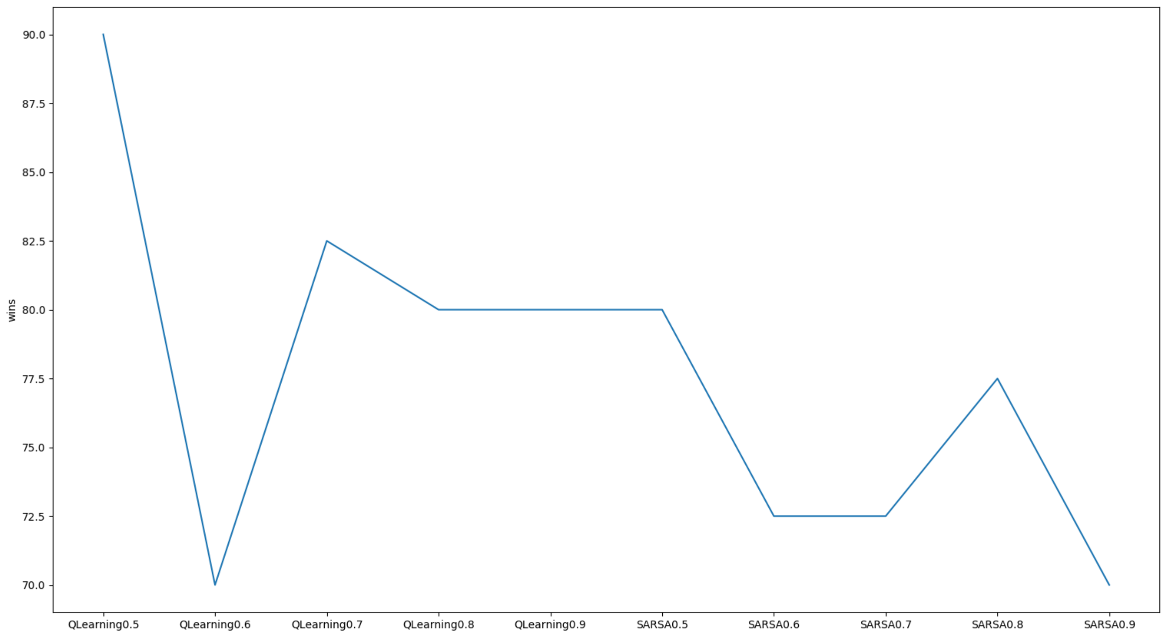


Figura 4.7: Percentage of wins for both algorithms with reduced range of gamma

For my surprise, the algorithm performed way better than expected for both `0.5` instances of the two algorithms, which left me wondering if this tendency would continue the lower we go with the parameter `gamma`. Luckily, Mario's game is way quicker to train and to get results from so I was able to train another range of the parameter (from 0.0 to 0.4) and test it to see if this tendency continues. These are the results:

4.2.1. Parameter Testing

4.2.1.0.1 SARSA

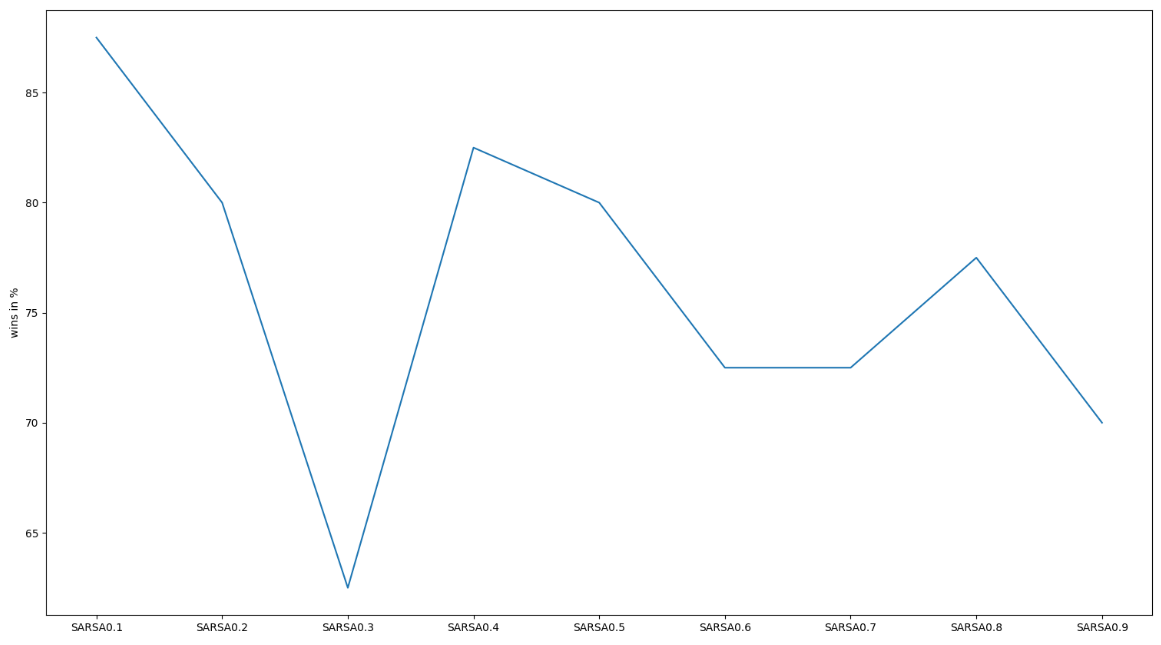


Figura 4.8: Percentatge of wins for SARSA algorithm

Surprisingly, our algorithm performed better with lower values of gamma, dropping a little consistency on instance `SARSA0.3`, which later skyrocketed to an amazing over 85 % win rate. The other's execution was quite good, averaging a 75 % win rate across instances.

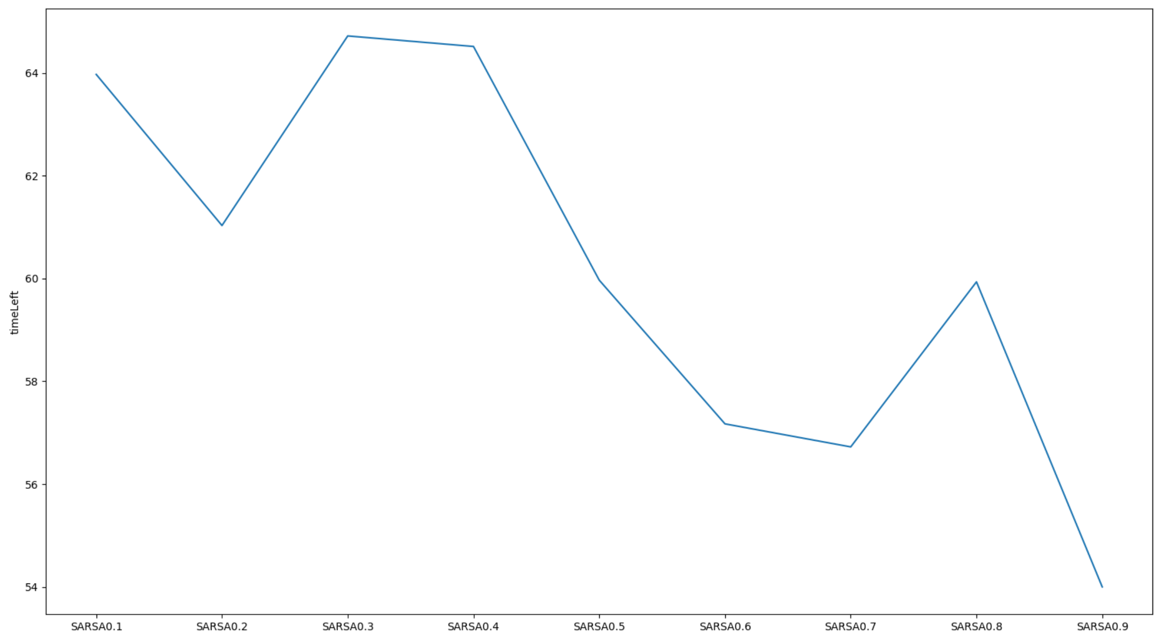


Figura 4.9: Averaged Time Left for SARSA

Here we have another metric, which purpose is to give us the opportunity to measure the the quality of our agent for this game. These are the averaged time of those executions in which mario finished the game. Thanks to this, we can tell that `SARSA01` actually is one of the fastest instances to finish the level, meaning that it is an agent that not only learnt his way through the level but it tried to completed fast.

4.2.1.0.2 Q-Learning

Now comes the turn to compare how well Q-learning, the apparently one of the most used techniques in reinforcement learning, did.

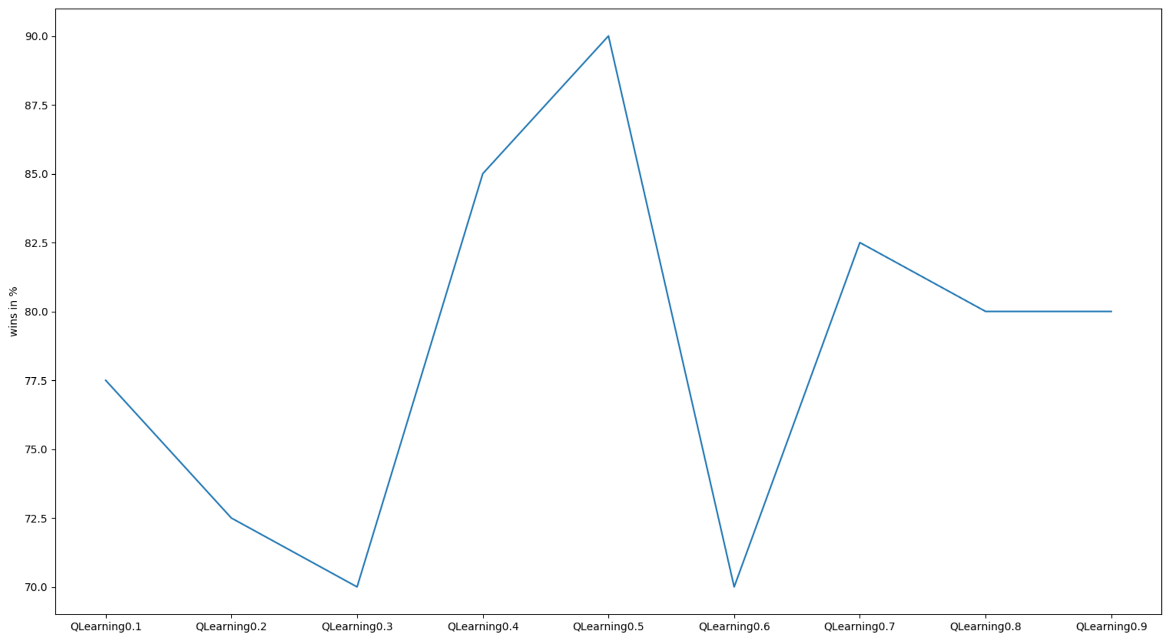


Figura 4.10: Percentage of wins for Q-Learning algorithm

In this case, `Q-Learning` instances didn't follow the tendency that the reduced range of parameters showed since it didn't improve going further down the parameter range. However, we can see an amazing 90 % win rate for parameter `gamma` 0.5 and two consistency drops around values 0.3 and 0.4 that might be caused for the fact that since it has such reduced vision of the future, it cannot have a good perception of how bad it is to hit an enemy.

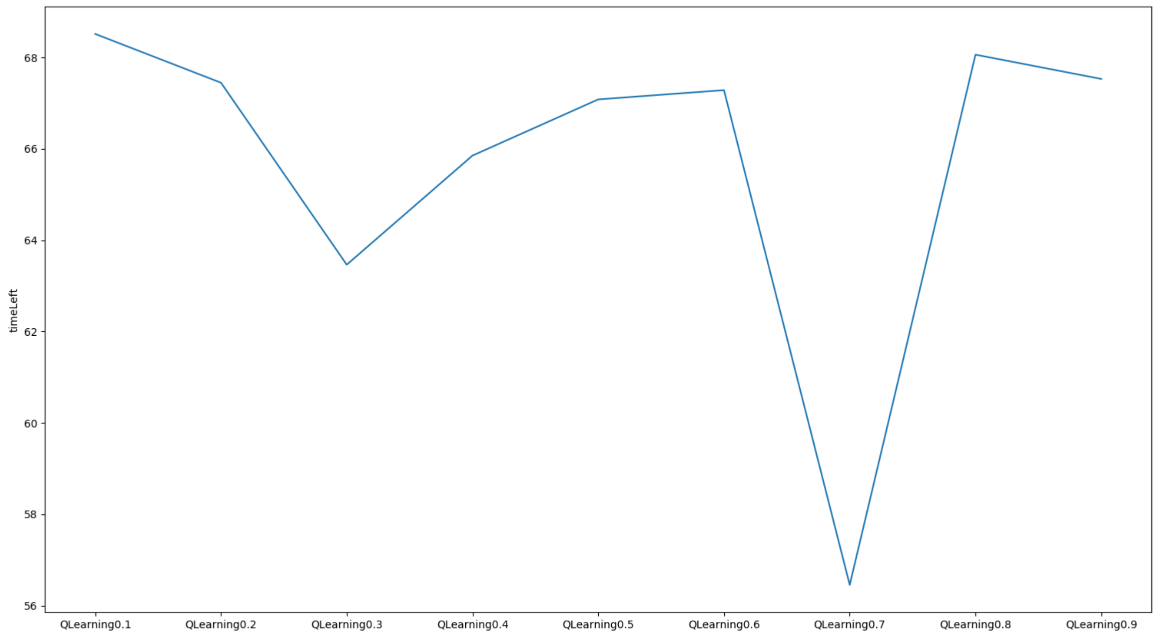


Figura 4.11: Averaged Time Left for Q-Learning

Once more we are looking to the averaged time left and we can appreciate how again, the instance which had the most win rate, also had a considerably high time left ater succesfully finishing the level. Also note how we could potentially get a few more seconds more if we consider using instance `Q-Learning0.1` sacrificing consistency of the agent.

4.2.2. Algorithm Testing

Let's now compare how both algorithms performed.

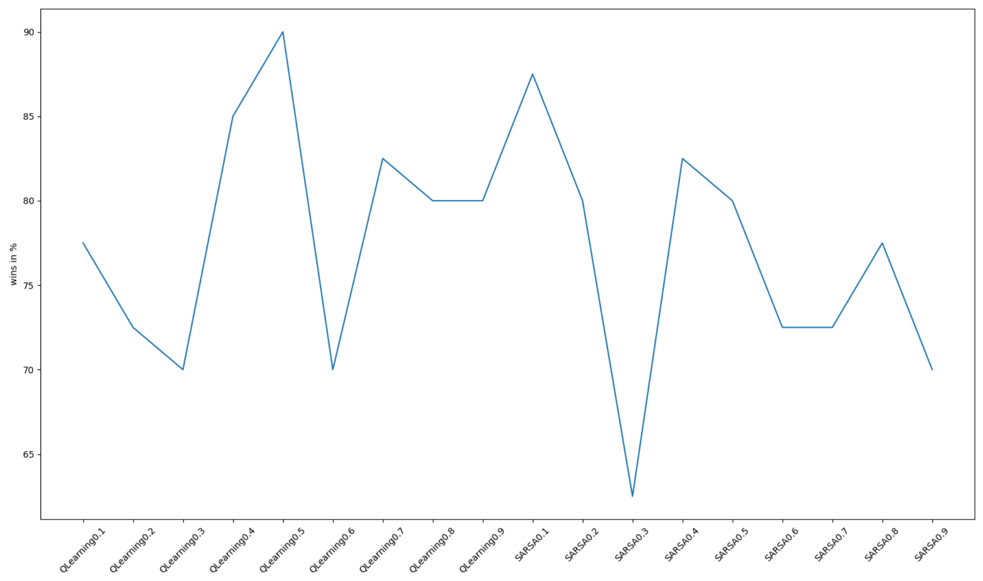


Figura 4.12: Percentatge of wins among algorithms

As we can tell, both algorithms perform very well across all instances, except `SARSA0.3` which drops to 63 % win rate. The two highest values are for `Q-Learning0.5`, with 90 % win rate, followed close by `SARSA0.1`, with 87 %.

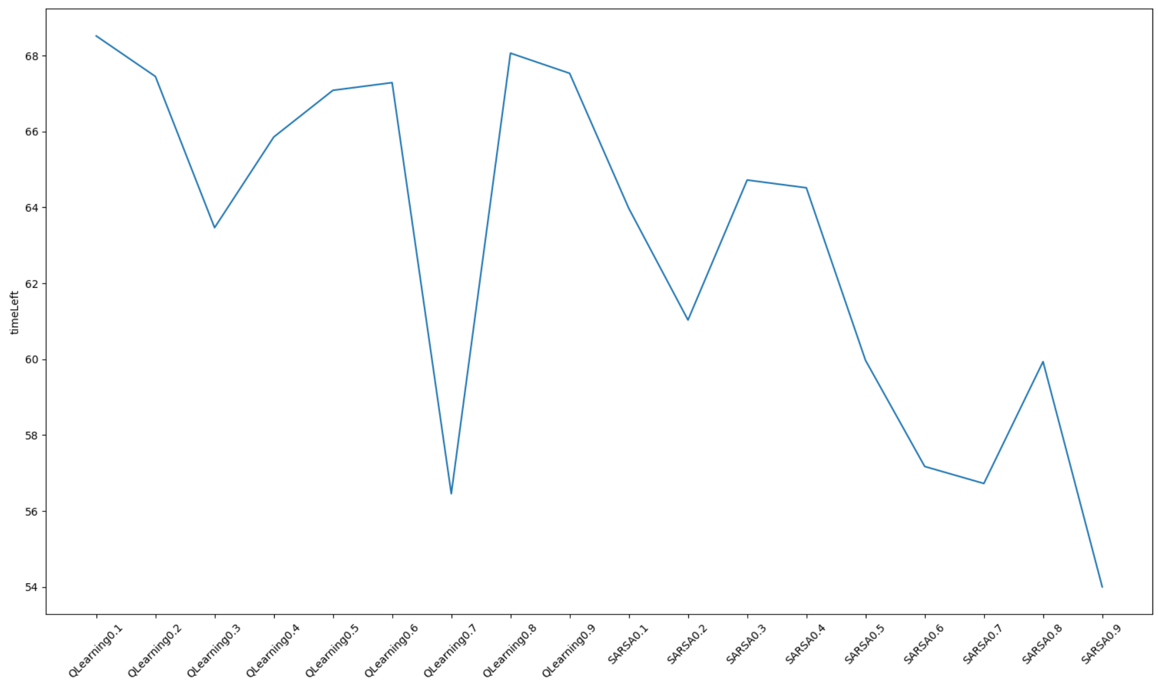


Figura 4.13: Averaged Time Left among algorithms

Here we can see a clear tendency of the Q-Learning instances completing the level slightly faster than the instances from `SARSA`. Meaning that `Q-Learning` not only has the most consistent agent, but also the fastest. These results actually makes sense for the kind of function updates that Q-Learning does, this topic will be further discussed in the conclusions section, when we are talking about the conclusion about the behaviour of both algorithms.

Capítulo 5

Conclusions

In this last section of my document, I will be talking about things that I learned about `Reinforcement Learning` and further explaining the results of the experiments, following this schema:

First, I will be talking about each game in particular,which set of algorithm and parameters work best and if we could potentially give an explanation on why the agents behave that way in the suggested game.

Then I will focus my attention to the algorithms, we'll be discussing topics like, how well did the algorithm performed on both games, which appears to be each flaw / strength...

After that, we'll be discussing how the different values for the parameter affected the performance of our agent and whether or not we can give a good intuition on which parameters would work best for other similar problems before hand.

Lastly, I will give some ideas of what I think about `Reinforcement Learning` and the algorithms after having to deal with its problems and virtues for the past 4 months, hopefully this last section will help someone who wants to dive in the topic a better understanding of the algorithms and what to focus to successfully train an agent.

5.1. FightingGameICE

This game cost a lot of time and resources to train, that is why I think the performance of the agents produced for this game is so low. I seriously expected more out of this game but its complex programming and not `Machine Learning` oriented made so much more difficult to implement a good agent for it. Also this game is really complex by itself, it has a lot of states and it becomes hard for the agent to visit them all in such low amount of time.

Anyways, even though we don't have an outstanding performance I am pretty happy that, for all the difficulties that the game presented at the beginning, it actually produced a good agent that is, at least, better than a randomly-driven agent. Sure it wouldn't be at the top of the competition, but a 62 % win rate with **Q-Learning04** is a pretty good result for the limitations. Keep in mind that these agents were trained for only 600 rounds which, compared to the `various` days of training that are being done for other agents of `Reinforcement Learning`, is a pretty low amount of training.Also keep in mind that the `Neural Network` used for this agent was pretty small, since the computer couldn't handle much bigger ones due to the lack of memory.

The results then, **does not** have to be seen as exact results of the performance of this algorithm in this game, rather than a tendency of what could potentially work.

It appears that for this kind of game there are two kind of strategies to follow in order to make the agent work with the current `reward system`.

- Give great importance to inmediate rewards
- Give great importance to potential rewards that will come in 10-15 states time

To follow the first strategy, we should give a really low value to our gamma. One good value would be **0.1**, as seen in **Figure** 4.5. For the second strategy it would be better to give intermidate values of gamma to our algorithm, for example **0.4** also seen in **Figure** 4.5. These values are consistent for both algorithms, that's why I think it's a safe assumption to make.

Keep in mind tough, this is representative for our current `Reward System` if we were to change it and, for example, give our agent a good enough amount of reward for winning the game, we would give him a further objective to look into and thus, we should be needing higher values of `gamma` to propagate this reward through our states.

Even though we have similarities on the behaviour of the parameter gamma, I would recommend to use Q-Learning over `SARSA` for this kind of game, since it appears to have a much better performance.

5.2. Mario

This game was quite the opposite from the `FightingGameICE`. The mario game version that I found [**MarioGame**] was really orientated to `Machine Learning` giving you full control over the environment, making it really easy to monitorize the performance of the agent at any given time. Also this game is much much simpler than the previous one, which benefits this kinds of "quick trainings", in the sense that the produced agents will be better at the game the simpler this is.

The results for this game were pretty surprising to me since I did not expect such performance in the first place. First of all, I expected parameters 0.9 to perform much much better than they did and I also expected that the range **0.5-0.9** would be more than enough to figure out how the algorithms performed in this kind of scenario.

Luckily, since the game was really well done so I was able to quickly increase the range of parameters to be used and train the agents for the new parameters in a relatively low amount of time. I say it is well done because the execution time on this game was extremely good, I did not suffer at any moment of long and broken executions as I did experience in the past game, thanks to the low quality textures and the simplicity of the game.

For this game we actually found different tendencies for the `gamma` values with the different algorithms. This is actually really interesting and help us see which is the difference on between these two and why `Q-Learning` is a much more popular algorithm. I will be going more in-depth in this subject in the following section.

Regarding the gamma values that work for this game, I was genuinely surprised at the performance of the highest consistent agent `Q-Learning05`. The first thought was that I probably misunderstood the meaning of the parameter `gamma` but, on a second thought this result actually makes sense. To understand it, we need to look at our `Reward System Policy`, If we give a closer look at what we are rewarding our agent for, we can see that we are giving him **a lot** of rewards for `MOVING` to the right. This reward might not seem much comparing it to the victory reward value, but keep in mind that there are a lot of states that will give a positive reward for moving to the right, making the sum of them bigger than what you get for completing the level.

But don't get me wrong, completing the level is still a **great** deal for the agent, because if he doesn't he will be severely punished. That is why we don't see that much consistency with low gamma values, because he still cares about what will be the following steps that will lead him to victory.

It is also fair to say that I could fit a much bigger `Neural Network` for this game thanks to the low use of resources it had. Nevertheless, I still think it was too small for the problem and I think it would greatly improve with bigger, more complex neural networks.

5.3. Algorithms

This kind of experiments I did are really representative, in my opinion, of which is effectively the difference between `SARSA` and `Q-Learning`.

As stated at almost the begining of this document, the only difference between these two is that `SARSA` take the resultant action of the interaction and `Q-Learning` gets the maximum among the possibilities. This explains why `Q-Learning` performed better than `SARSA` on both games. However, this difference doesn't mean that SARSA is worse than Q-Learning by any means. It only goes to show that SARSA is **slower** on deciding which are the best actions than `Q-Learning`.

When you think about `Q-Learning` taking the **maximum** value function from all the possible states you can end up to from the specific state we tend to think that this is why Q-Learning should work better than SARSA. Truth is, that to select the action, you are using a $\epsilon - greedy$ policy meaning that, most of the time, you will be already be getting the maximum value function for the state-action you might end up to. This mean that in those cases, `SARSA` is actually behaving exactly the same way as `Q-Learning`. Note though, that I said **most of the time**, implying that it will take longer to `SARSA` to find the optimum actions to do but, just as Q-Learning does, it will get there.

And yet the question remains: Which is better then? The answer is very simple, it depends. Giving enough time they will both most likely be equal, if we give a greedy enough policy. Then the question should go more like: Which will give me results faster? And the answer again is, it depends. And the thing is that it doesn't only depend on the `problem` but also the situation. If we use `Q-Learning` it always uses the maximum of the possible states from what he knows **for the moment**. That means that it could enter some state where he thinks he is doing what he must do, but since all the time is getting the maximum, it doesn't know if there's something better. In this case, with SARSA it could manage to get around and visit all states by itself.

Even though all the things I discussed, I would suggest to have `Q-Learning` for the way to go. The problems that I listed in the previous paragraph are actually solved by using a good enough policy which will force us to visit all states, rather than just the optimal ones. In our case, $\epsilon - random$ is a well enough choice. What I meant by the statements before is that the algorithm by itself suffers from being too greedy and will not explore by itself, that is why we need that exploration to be explicit.

5.4. Parameter Gamma

Because of the project limitations, I couldn't have a more representative set of experiments to see how the agent's performance is affected by the parameter values.

Even though I can't give full proof of what this parameter represents, I found that my results follow more or less the intuition given in [**Sutton**]'s book that this value means how much we care about future rewards. We can see this kind of behaviour in both of the games but not clearly enough to confirm that this behaviour will always be true.

As a result I can't tell which parameters works best for each style of games, more than just have a general range of what could actually work, test it and then deciding.

5.5. General Thoughts

On a general note, I am pretty happy of how this project went and the things I learnt about `Reinforcement Learning` for such a limited amount of time. However, this is a really complex topic where too many factor affect the performance and the success of the agent.

For this reason, I have not enough data to make statements on which algorithms work best in certain situations with certain parameters. Nevertheless, as I stated earlier, it was well enough to see the tendency of the parameters and its parameters for the designated games.

If I were to recommend the usage of one of the two algorithms, I would definitely recommend `Q-Learning` over `SARSA` since it seems to have a better performance. Regarding parameters, I would focus more on how the `Reward System` is built and then decide from there. If the Reward system is built to make immediate rewards more important than the ones in the long run, then use low values of `gamma`, otherwise if it is designed to reward more on the long run use high values of `gamma`. If we focus on the reward system rather on the game, we will get much better accuracy when selecting the range of values, and should avoid surprises like the one I had with `Mario`.

Regarding if it is possible to create an agent that plays the game without having to worry too much about the game, I cannot ensure that it is but I can't ensure it can be done either. The problem to make such strong statement is what the definition of "having to worry too much about the gameï. If we focus on Mario's agent, I only had to provide some basic information about general knowledge about how the game works: you are going to the right, you are doing `fine`. This approach actually gave an agent that plays pretty decently a level of Mario. On the other hand we have `FightingGameICE` which again, we gave very little information about the game mechanics: You hit, you are doing `fine`, you are being hit, you are doing `badly`. Yet this approach didn't gave us a really good agent at all.

If we follow this kind of intuition, we could say that this statement heavily depends on the complexity of the game we are trying to make the agent for. However, more games should be tried to actually proof this theory.

Capítulo 6

Future work

This project can be extended and continued in many many directions, some of them are:

- **Add more games** I've been able to make a general enough algorithm, that will be able to be used for any other game with very little changes.
- **Work on neural networks** I think that the agents could improve **a lot** if the neural networks get a little more attention and improvement.
- **Train harder** The experiments could be extended with more range of parameters and more training time that, for time issues, I wasn't able to do.
- **Reward System Changes** I think that it would be really interesting to see how different configurations of the reward system affect to the executions of the trained agents for the different parameter values.

Capítulo 7

Bibliography

[1] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* Online-Draft of Second the Edition,2014-2017. `http://incompleteideas.net/sutton/book/bookdraft2017june.pdf`

[2] hris Woodford. *Neural Networks.* On-line article, 2017. `http://www.explainthatstuff.com/introduction-to-neural-networks.html`

[3] Sergey Karakovskiy and Julian Togelius. *Mario AI Competition.* Official Mario AI Competition page, 2010. `http://julian.togelius.com/mariocompetition2009/`

[4] INTELLIGENT COMPUTER ENTERTAINMENT LAB., RITSUMEIKAN UNIVERSITY. *FightingGameICE.* Official FightingGameICE page, 2017. `http://www.ice.ci.ritsumei.ac.jp/~ftgaic/index.htm`

[5] David Bigas Ortega. *Machine Learning Applied to Pac-Man Final Report.* Final Project from UPC, 2015. `http://upcommons.upc.edu/bitstream/handle/2099.1/26448/108745.pdf?sequence=1`

[6] Purvag Patel. *Improving Computer Game Bots' behavior using Q-Learning.* thesis from Southern Illinois University Carbondale,, 2009. `http://opensiuc.lib.siu.edu/cgi/viewcontent.cgi?article=1083&context=theses`

[7] David Silver. *UCL Course on RL.* Course with video-lectures, 2015. `http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html`

[8] Alexander Jung. *Playing Mario with Deep Reinforcement Learning.* Source Code for a Reinforcement Learning project of Mario, 2016. `https://github.com/aleju/mario-ai`

[9] Community. *Deep Learning Library page.* official page for the library used for neural networks, 2017. `https://deeplearning4j.org/`

[10] Akshay Srivatsan. *DeepLearningVideoGames.* Source code for a Reinforcement Learning project, 2016. `https://github.com/asrivat1/DeepLearningVideoGames`

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller. *Playing Atari with Deep Reinforcement Learning.* Paper, 2016. `https://arxiv.org/abs/1312.5602`

[12] Ian Goodfellow and Yoshua Bengio and Aaron Courville. *Deep Learning.* MIT Press, 2016. `http://www.deeplearningbook.org/`

[13] Alex Osés. *Reinforcement Learning in videogames.* Source code for this project, 2017. `https://github.com/Outer2g/Reinforcement-Learning-Project`

Capítulo 8

Appendix A: FightingGameICE Results

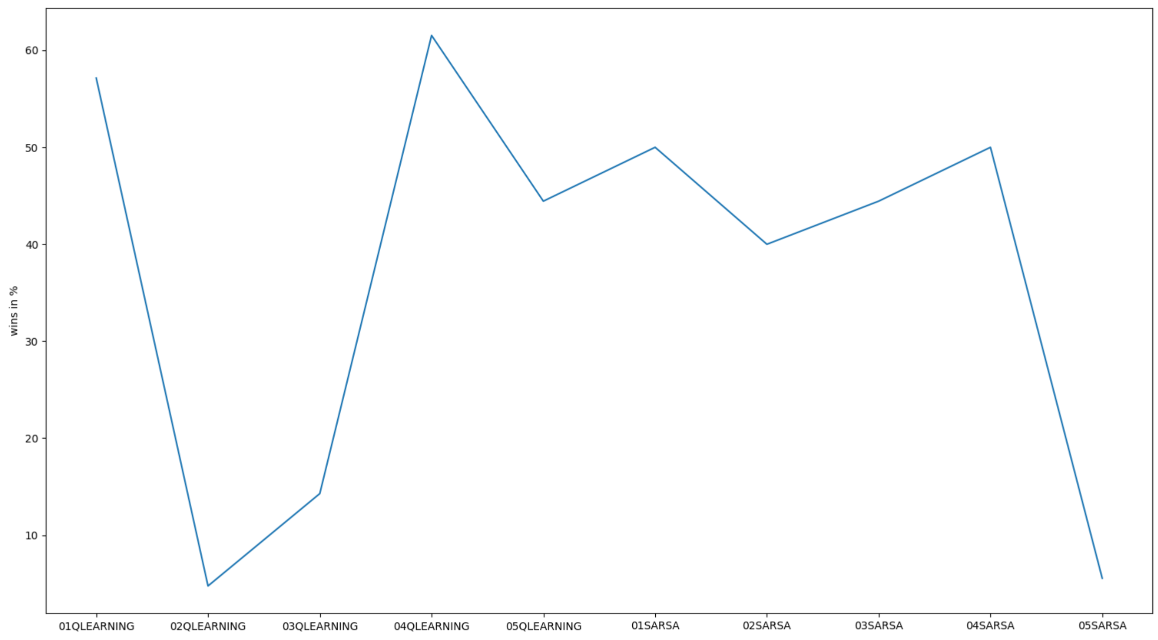


Figura 8.1: Percentage of wins from both algorithms in FightingGameICE

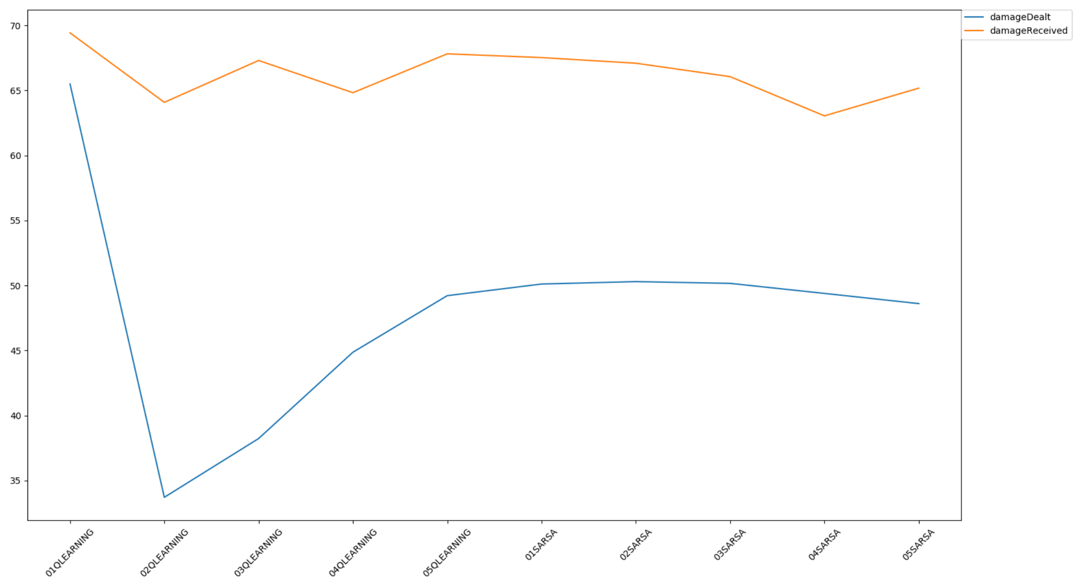


Figura 8.2: Damage Dealt vs Damage Received comparison from both algorithms in FightingGameICE

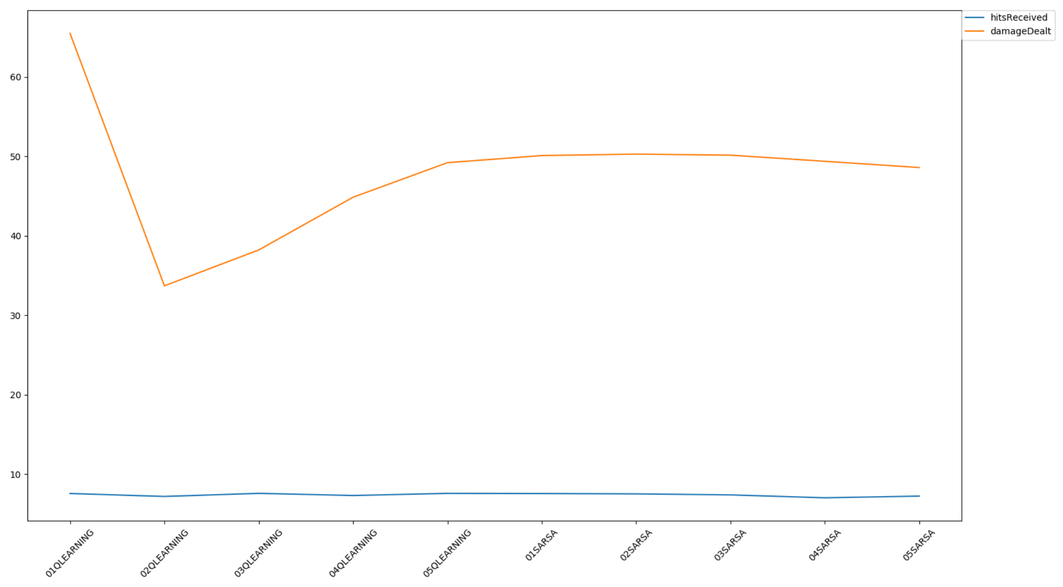


Figura 8.3: Hits Dealt vs Hits Received comparison from both algorithms in FightingGameICE

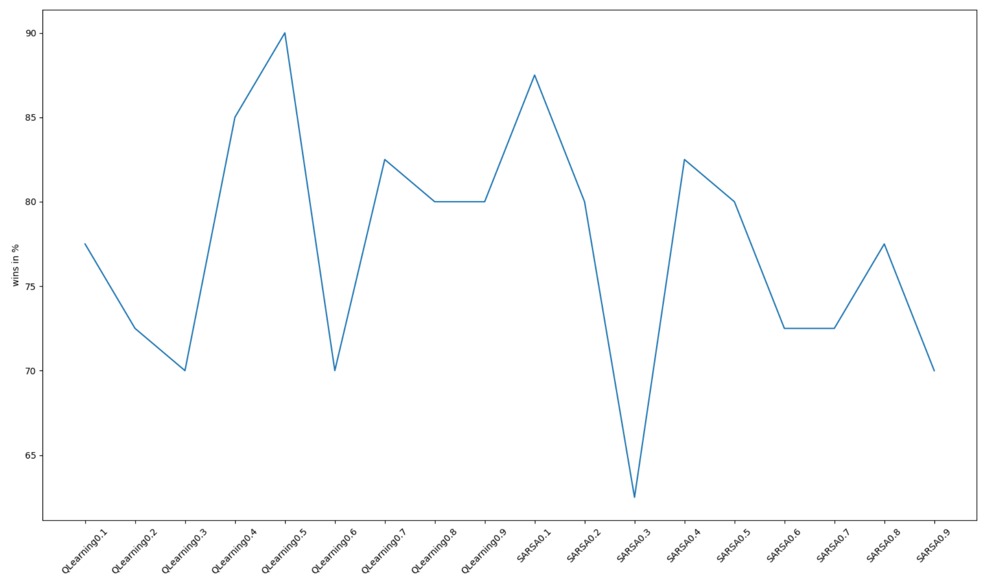


Figura 8.4: Percentage of wins from both algorithms in Mario

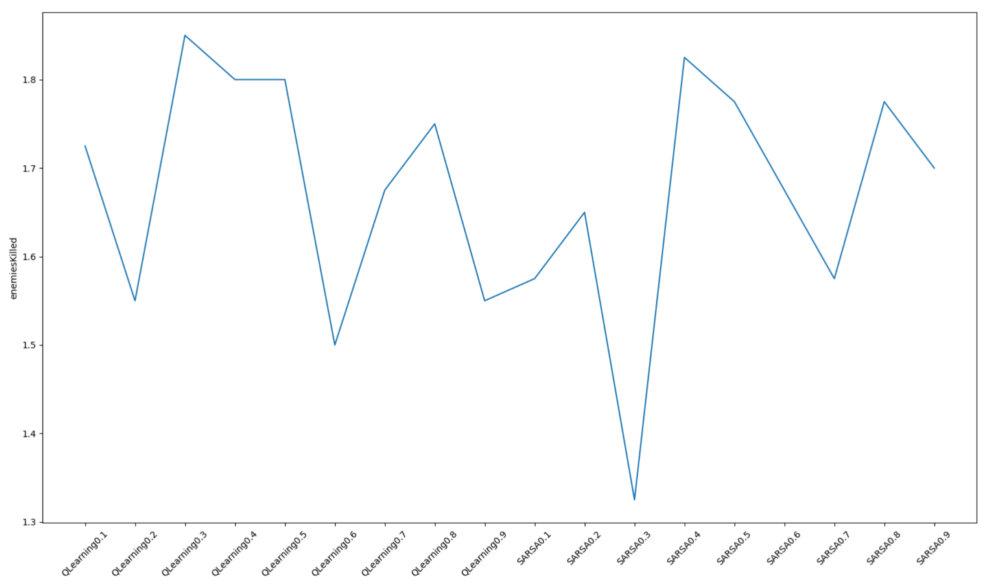


Figura 8.5: Enemies killed from both algorithms in Mario

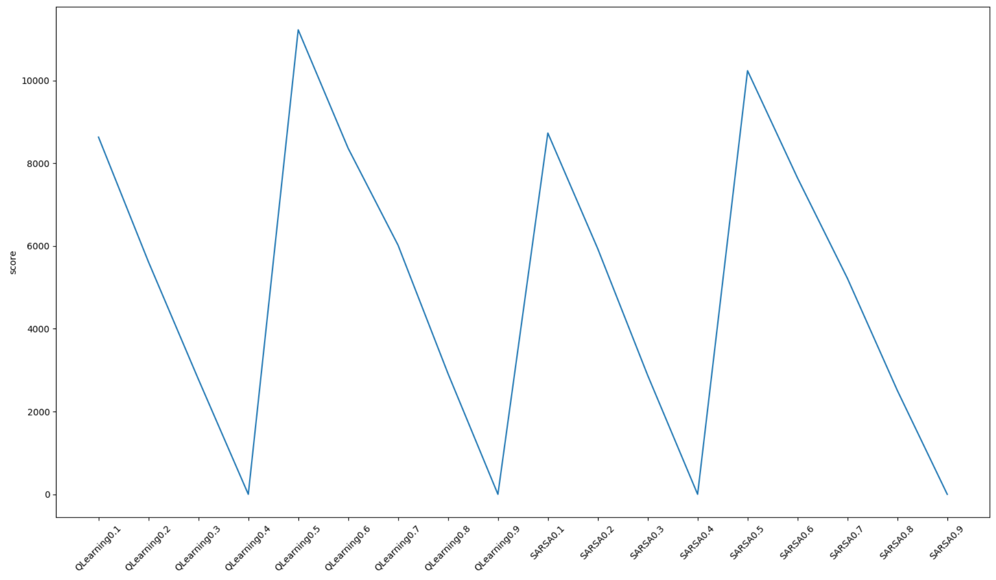


Figura 8.6: Global Score from both algorithms in Mario

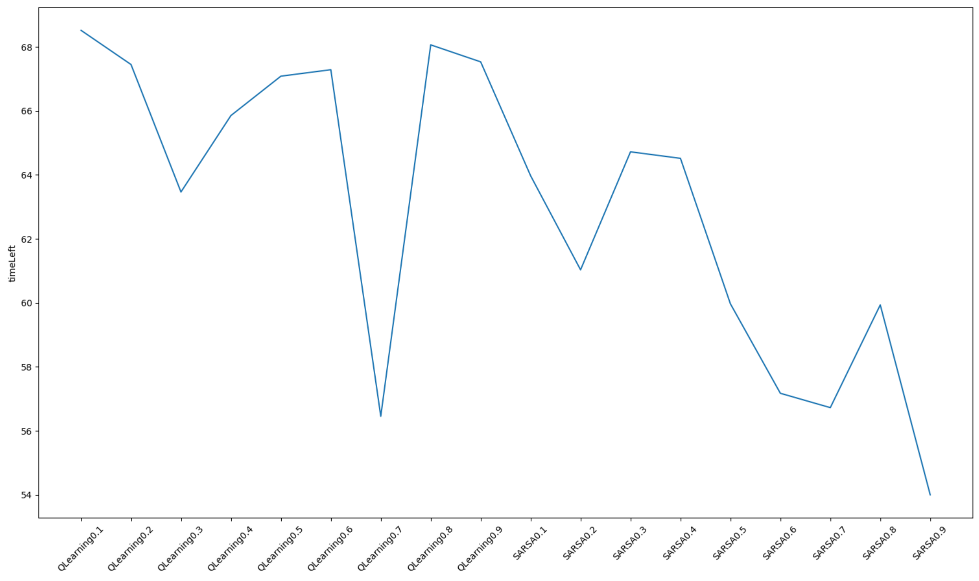


Figura 8.7: Averaged time left at the end of the level from both algorithms in Mario