Reinforcement learning in videogames

Raúl Villar Casino

February 6, 2024

Resum– Aquest treball pretén aprofundir en els principals models d'aprenentatge per reforç (RL) i explorar el seu potencial en entorns de diversa complexitat. Com a punt de partida, es va realitzar una revisió exhaustiva de l'estat de l'art, abastant tant mètodes tabulars (Q-Learning, Value Iteration, MonteCarlo) com Deep RL (Deep Q-Learning, PPO). Posteriorment, es van implementar els mètodes tabulars al joc FrozenLake amb l'objectiu de comparar el seu funcionament en un entorn senzill i identificar el més eficaç, resultant Value Iteration com l'opció òptima en aquest context. Finalment, es van entrenar models DRL per a jugar a Breakout a partir de captures de pantalla, comparant Deep Q-Learning i PPO. En aquest cas, PPO va demostrar un rendiment superior, consolidant-se com una opció potent per a l'entrenament d'agents en entorns basats en imatges.

Paraules clau– Aprenentatge per Reforç, Mètodes Tabulars, Solucions Aproximades, Q-Learning, Deep Q-Learning, Value Iteration, Mètodes Montecarlo, PPO, Breakout, Frozenlake, Gymnasium

Abstract– This work aims to delve into the main Reinforcement Learning (RL) models and explore their potential in environments of varying complexity. As a starting point, an exhaustive review of the state-of-the-art was conducted, covering both tabular methods (Q-Learning, Value Iteration, Monte Carlo) and Deep RL (Deep Q-Learning, PPO). Afterwards, the tabular methods were implemented in the FrozenLake game with the aim of comparing their performance in a simple environment and identifying the most effective one, resulting in Value Iteration as the optimal option in this context. Finally, DRL models were trained to play Breakout from screenshots, comparing Deep Q-Learning and PPO. In this case, PPO showed superior performance, consolidating itself as a powerful option for training agents in image-based environments.

Keywords– Reinforcement Learning, Tabular Methods, Approximate Solutions, Q-Learning, Deep Q-Learning, Value Iteration, MonteCarlo, PPO, Breakout, Frozenlake, Gymnasium

1 INTRODUCTION

Reinforcement learning (RL), inspired by behavioral psychology and the idea of trial-and-error learning, introduced a novel approach to Artificial Intelligence (AI). Unlike supervised learning, where models are trained on labeled data, and unsupervised learning, which focuses on discovering patterns within data, Reinforcement Learning agents learn by interacting with an environment and receiving feedback in the form of rewards. This paradigm shift led to the creation of intelligent agents capable of making sequential decisions and adapting to dynamic environments [1].

One of the early milestones in Reinforcement Learning was the development of Q-learning by Watkins in 1989 [2], which laid the foundation for value-based methods. Subsequent years saw the emergence of deep Reinforcement Learning, with algorithms like Deep Q-Networks (DQNs) and policy gradients, resulting in remarkable achievements in gaming and robotics. The victory of AlphaGo [3] over the world champion in the complex board game of Go in 2016 [4] marked a watershed moment, demonstrating Reinforcement Learning's prowess in mastering intricate tasks.

Today, Reinforcement Learning is at the forefront of AI research, with applications spanning from autonomous vehicles and healthcare to finance and natural language processing. Its key differentiator lies in its ability to learn from experience, adapt to uncertainty, and optimize decisions in dynamic and complex environments.

This project falls within the realm of Artificial Intelligence and Reinforcement Learning, two areas of research and development that have seen significant growth in recent years. Reinforcement learning is one of three basic machine learning paradigms. It is about learning the optimal behavior in an environment to obtain maximum reward. In Rein-

• Contact e-mail: raul.escacs@gmail.com
• Specialization: Computing
• Work tutored by: Jordi Casas Roma (Computing)
• Course 2023/24

forcement Learning, the data is accumulated from machine learning systems that use a trial-and-error method. Data is not part of the input that we would find in supervised or unsupervised machine learning.

Reinforcement learning is a machine learning technique inspired by how humans and other organisms learn through interaction with their environment. It is particularly relevant in applications where an agent must make sequential decisions to maximize cumulative reward, such as in robot control, gaming, and recommendation systems [5].

Reinforcement learning uses algorithms that learn from outcomes and decide which action to take next. After each action, the algorithm receives feedback that helps it determine whether the choice it made was correct, neutral or incorrect. It is a good technique to use for automated systems that have to make a lot of small decisions without human guidance.

2 Objectives

The primary goal of this project is to effectively deploy various Reinforcement Learning algorithms across a range of video games, each increasing in complexity. Subsequently, we aim to assess their performance and determine which algorithm excels in achieving specific objectives.

To achieve this overarching objective and facilitate a comprehensive algorithmic comparison, we have segmented our approach into the following key components:

1. In-depth exploration of Reinforcement Learning theory, encompassing a thorough examination of the models. This exploration will encompass an analysis of their architectural aspects as well as the salient characteristics that distinguish these models.
2. Implementation of the proposed models using the Python programming language, leveraging the Gymnasium [6] library developed by *The Farama Foundation*.
3. Development of a suite of functionalities designed to visualize the outcomes generated by these models, enabling us to make meaningful comparisons among them.
4. Systematic experimentation with the hyperparameters of the models, with the aim of gaining a deeper insight into their workings and optimizing their performance to achieve the best possible results.

3 Methodology and Planning

For the execution of this project, we have opted for the utilization of the Kanban methodology [7]. This choice is rooted in its capacity to offer a straightforward and efficient means of visualizing project progression. The employment of Kanban boards provides clear insights into the status of individual tasks and delineates the subsequent actions required to propel the project forward.

Furthermore, Kanban facilitates the prioritization of tasks based on their significance and immediacy at any given juncture. This enables a concentration of effort on the most pivotal tasks pertinent to the project, while preventing an undue burden on lower-priority endeavors. This strategic approach aids in the optimization of both time and resources, thereby enhancing the prospects of attaining the most favorable outcomes.

To operationalize the Kanban methodology within this project, we have elected to implement columns, including *Backlog*, *To Do*, *In Progress*, *Done*, and *Blocked*.

The Backlog column serves as the initial repository for all pending tasks. Subsequently, the *To Do* column accommodates the most critical tasks for the current project phase and establishes those earmarked for completion during the sprint.

Tasks underway within the ongoing sprint find their place in the *In Progress* column and are subsequently moved to the *Done* column upon completion. The presence of a *Blocked* column allows for the identification of tasks impeded by various factors, such as resource constraints or the need for additional information.

To enhance task and time management, we have divided the project into two-week sprints. This approach affords the opportunity to assess project progress periodically and adapt planning as needed, guided by the objectives achieved to date.

To implement this methodology, we will employ a Kanban Board within the web-based Asana platform [8]. This Kanban Board will be complemented by the utilization of a Gantt chart to delineate the tasks for each sprint.

The project will be divided into two primary segments:

1. Development of Reinforcement Learning in the Frozen Lake Environment (a simpler environment):
   - **2 weeks**: Comprehensive study and application of the MonteCarlo algorithm
   - **2 weeks**: In-depth exploration and application of Dynamic Programming
   - **2 weeks**: Mastery and application of the Q-Learning algorithm
   - **1 week**: Comparative analysis and examination of the benchmarks achieved
2. Development of Reinforcement Learning in the Breakout Environment (a more complex environment):
   - **3 weeks**: Proficiency and application of Deep Q-Networks
   - **3 weeks**: Competence and implementation of Policy Gradients methods
   - **2 weeks**: Thorough assessment and scrutiny of the benchmarks obtained

Additionally, one week is allocated for documentation purposes, and an extra week remains unassigned to cater to any unforeseen delays or tasks that may require more time than initially anticipated. For a comprehensive view of the Gantt chart and associated tasks, please refer to Section A.1.

4 Reinforcement Learning Basics

In this section, we will go through the basic concepts that form the fundamentals of Reinforcement Learning, in order to understand the mechanisms making intelligent decision-making in dynamic environments.

Unlike traditional programming, where specific instructions are provided, Reinforcement Learning algorithms learn autonomously. They are not explicitly told what to do. Instead, they discover optimal behavior through their own exploration and feedback from the environment.

An agent initially lacks knowledge of the environment, but it receives observations and takes actions randomly. Following an action, the environment provides a new observation and a reward, indicating the value of that action. Through repeated interactions and feedback, the agent develops an optimal policy for playing the game.

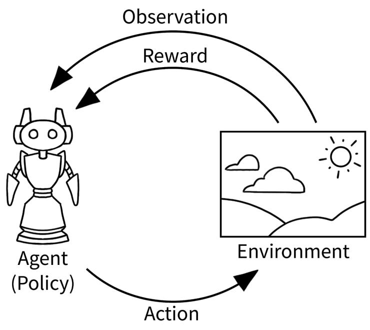


Fig. 1: Reinforcement learning Scheme

4.1 Preliminaries

First, we will explain some key terms:

- **State**: Description of the situation the agent is in his environment. In a game, it can be a screenshot of what is happening on the screen.
- **Action**: A choice that the agent can make in a state. In a game, can be a move, or any other input the player can do.
- **Value function**: A function that assigns a value for each state, in order to assign how good is to be in that state.
- **Bellman Equation [9]**: An equation that relates the value of a state to the value of the next states. It is a fundamental equation in Reinforcement Learning.
- **Policy**: A function that assigns an action to each state.

4.2 Markov Decision Process (MDP)

In mathematics, a Markov decision process (MDP) [10] is a discrete-time stochastic control process. It provides a mathematical framework for modeling decision-making in situations where outcomes are partly random and partly under the control of a decision maker. MDPs are useful for studying optimization problems solved via dynamic programming. They are used in many disciplines, including robotics, automatic control, economics and manufacturing.

The key defining feature is the Markov property, asserting that the future state of the system depends solely on its current state and the action taken, regardless of the path that led to the current state. In Reinforcement Learning, MDPs encapsulate the dynamics of the agent-environment interaction, comprising states, actions, transition probabilities, and rewards.

4.3 Tabular and Approximate Methods

Reinforcement Learning methods can be broadly categorized into two main models:

Tabular Methods:

- These methods explicitly store and update a table to represent the value function or policy.
- Tabular methods are effective in scenarios with a manageable number of states and actions.
- Q-learning and MonteCarlo methods are examples of tabular approaches.

Approximate Methods:

- Approximate methods generalize learning across a continuous or large state and action space without explicitly enumerating all possibilities.
- These methods leverage function approximation techniques, such as neural networks, to handle complex and high-dimensional state spaces.
- Deep Q-Networks (DQN), Policy Gradients, and Actor-Critic architectures are examples of approximate methods.

5 State of the Art

Before we start with the experiments and results, we will explain what methods and environment we are going to use.

Our environments will be provided by Gymnasium [6]. Gymnasium is an open source Python library for developing and comparing Reinforcement Learning algorithms by providing a standard API to communicate between learning algorithms and environments, as well as a standard set of environments compliant with that API.

Our agents (algorithms) will be coded by us, reimplementing some famous algorithms that have been created in Reinforcement Learning history.

The environments provided by Gymnasium have a lot of useful functions and classes to create and analyze Reinforcement Learning algorithms. We will focus on the most important ones:

- **Action Space**: Every environment specifies the format of valid actions and observations. This is helpful for both knowing the expected input and output of the environment, as all valid actions and observation should be contained with the respective space. If an environment only accepts the inputs for moving up, down, left and right, the action space will be a range from 0 to 3, in which every number indicates one action.
- **Observation Space**: It is what the agent will see, the observation can be different things for different environments. The most common form is a screenshot of the game. There can be other forms of observations as well, such as certain characteristics of the environment described in vector form [11].

- **Rewards**: The reward that you can get from the environment after executing the action that was given.
- **Step()**: It is the main function to make our environment work. Updates an environment with the given action, returning the next agent observation, the reward for taking that actions and if the environment has terminated or truncated due to the latest action.

For this thesis, we will compare some basic Reinforcement Learning algorithms and complex Reinforcement Learning with neural networks. Simple ones won't work on a more complex (with more states, actions and observations) environment. So we will split the study in two environments.

6 Tabular methods

6.1 Models

In this section, we will explore in detail the algorithms that have been instrumental in the execution of our project. Algorithms are the pillar in Reinforcement Learning and serve as the engines that power our agent's ability to make intelligent decisions in dynamic and challenging environments.

By comprehending the underlying mechanics of these algorithms, we will be able to analyze and understand the results of our work and comprehend how they have approached the challenges presented in the gaming environment. As we progress in the exposition of these algorithms, we will appreciate their influence on intelligent decision-making and their potential for applications beyond our current project.

First one will be Q-Learning [2] algorithm. Q-Learning is a Reinforcement Learning algorithm used to train an agent to make optimal decisions in an environment it interacts with. The goal is to learn a value function called "Q" that estimates the expected value of taking a particular action in a specific state and following an optimal policy.

Q-Learning doesn't know anything about the environment, and learns with a trial and error method. The pseudocode is presented in Algorithm 1.

- The explorer policy in line 5 takes a random number between 0 and 1, and compares it to epsilon. If the random number is smaller we make a random action (exploration), if not we take the action with highest Q-Value for the actual state (exploitation).
- When we take an action in line 6. We use the method *step()*, which receives our action and returns: new state, reward and if the game has terminated or truncated.
- In line 7, the elements of the equation are:
  - $\boldsymbol{\alpha}$: Learning rate.
  - $\boldsymbol{Q(s_t, a_t)}$: Current value.
  - $\boldsymbol{r_t}$: Reward.
  - γ: Discount factor.
  - $\boldsymbol{maxQ(s_{t+1}, a)}$: Estimate of optimal future value

Next one will be Value Iteration [12]. Value Iteration is a dynamic programming algorithm used to solve Markov Decision Process (MDP) problems and find the optimal policy for an agent in that environment. It is simpler than Q-Learning, but is more limited to stochastic environments.

The objective is to find an optimal policy, which is a strategy that determines which action to take in each state to maximize the cumulative reward over time. Value Iteration is based on the concept of the value function. For each state, the value function, denoted as V(s), represents the expected cumulative reward that the agent can obtain if it starts in that state and follows a specific policy. The pseudocode is presented in Algorithm 2.

- In line 6, the elements of the equation are:
  - $\boldsymbol{P(s, a, s')}$: Probability of going to state s' from state s and action a.
  - $\boldsymbol{R(s')}$: Reward of going to state s' from state s and action a.
  - γ: Reward corrector.
  - $\boldsymbol{V(s')}$: Estimated value of next state s' with the actual policy.
- ϵ is a very small positive threshold.

Last one is MonteCarlo Algorithm [13]. MonteCarlo methods are ways to solve the Reinforcement Learning problem using a very simple idea: the average is a good estimator of the expected value. Therefore, they are based on estimating the value functions v_π(s) or q_π(s,a) from sample averages of the return for each state (or state-action pair)

With Montercarlo, learning from real experience does not require a priori knowledge of the dynamics of the environment to achieve optimal behaviors. And in simulated environments, although we need a model of the environment to generate state transitions, it is not necessary to know the exact expression of the probability distributions. The pseudocode is presented in Algorithm 3.

6.2 Environment

For tabular methods, we will use Frozenlake (Fig. 2). Frozenlake involves crossing a frozen lake from start to goal without falling into any holes by walking over the frozen lake. The player may not always move in the intended direction due to the slippery nature of the frozenlake.

The given action space will be a range from 0 to 3, which will let the player move up, down, left and right. And every time the player moves, it will get a reward based on his performance:

- **Ice**: The reward will be 0
- **Goal**: The reward will be 1
- **Hole**: The reward will be 0

With this reward, the main objective is to complete the game, but there won't be any penalty for doing extra steps or falling, due to don't have any negative reward.

Algorithm 1 Q-Learning

```
1: Initialize a Q-value table with arbitrary values for each pair (state, action).
2: for every episode do
3:     Restart environment and obtain initial state $s$
4:     while not done do
5:         Choose an action $a$ based on the explorer policy from Q-values
6:         Take action $a$ and observe the reward $r$ and the new state $s'$
7:         Update the Q-value of the state-action pair using the Q-value update equation:
8:             $Q^{new}(s_t, a_t) = (1 - \alpha) \cdot Q(s_t, a_t) + \alpha \cdot [r_t + \gamma \cdot maxQ(s_{t+1}, a)]$
9:         Update the current state $s \leftarrow s'$
10:    end while
11: end for
```

Algorithm 2 Value Iteration

```
1: Initialize the value function $V(s)$ arbitrarily for all states $s$
2: repeat
3:     $\Delta \leftarrow 0$
4:     for each state $s$ do
5:         $v \leftarrow V(s)$
6:         $V(s) \leftarrow \max_a \sum_{s'} P(s,a,s')[R(s,a,s') + \gamma \cdot V(s')]$
7:         $\Delta \leftarrow \max(\Delta, |V(s) - v|)$
8:     end for
9: until $\Delta < \epsilon$
```

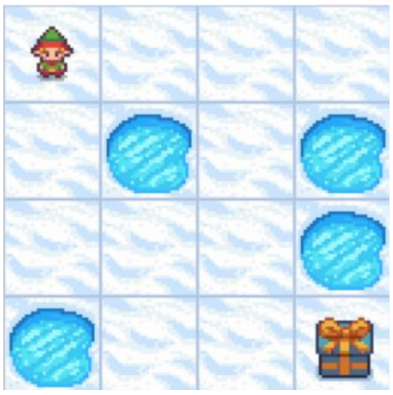


Fig. 2: Frozenlake screenshot

6.3 Results

In this section, we will discuss the results of the algorithms, and understand why they make this performance. The parameters we will use are:

- **Total episodes = 2000 (QL) / 10 (Value Iteration) / 70000 (MonteCarlo)**: Number of times the environment is going to be played for every run.
- **Total runs = 5**: Number of times we will reproduce the experiment to have more consistent results.
- **Frozen probability = 0.9**: Probability of a tile being frozen and not a hole.
- **Maximum number of steps = 250**: After an algorithm reaches the maximum, even if it hasn't finished, the environment stops.
- **Map sizes = [4,7,9,11]**: We will test the algorithms in different map sizes of NxN, where N is the number in the array.
- **Gamma (QL) = 0.95**: Discounting rate.
- **Gamma (Value iteration) = 1**: Reward corrector for next state estimated value.
- **Theta = $1 \cdot 10^{-8}$ (Value iteration)**: Very small number that is used to decide if the estimate has sufficiently converged to the true value function.
- **Epsilon = 0.1 (QL) / 0.01 (MonteCarlo)**: Exploration probability

Before we analyze every algorithm, let's take a look at the summary Table 1.

We can see that Value Iteration is clearly better for this task. This doesn't necessarily mean that Value Iteration is a better algorithm in general, just is a better algorithm for playing Frozenlake. MonteCarlo gets the second position by getting very good results, but with the highest time and energy usage.

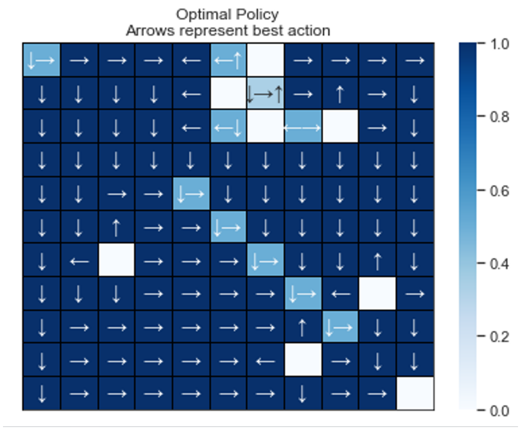


Fig. 3: Policy of Value Iteration

Algorithm 3 MonteCarlo epsilon soft Policy

```
Initialize:
$\pi \leftarrow$ Initialize $\epsilon$-soft policy
Initialize action-value function $Q(s, a)$ arbitrarily for all states and actions
Initialize visit count table $N(s, a)$ for all states and actions to zero
while not finished (for each episode) do
    Generate an episode following policy $\pi$: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    for Every step for every episode $t \leftarrow T - 1$ to 0 do
        $G \leftarrow \gamma G + R_{t+1}$
        if pair $(S_t, A_t)$ is the first visit in the episode then
            Update visit count: $N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$
            Update action-value: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{1}{N(S_t, A_t)}(G - Q(S_t, A_t))$
            Update $\epsilon$-soft policy $\pi$
        end if
    end for
end while
Return: $\pi_*$, the optimal $\epsilon$-soft policy
```

TABLE 1: FROZENLAKE ALGORITHMS PERFORMANCE

	Time (seconds)	Winrate (%)	Memory Usage (MiB)	Energy Usage (CO2 $\mu Kg/m^3$)
Value Iteration	3.76	100%	195.20	17.02
Q Learning	78.27	53%	266.50	336.41
MonteCarlo	538.32	93%	137.50	1518.14

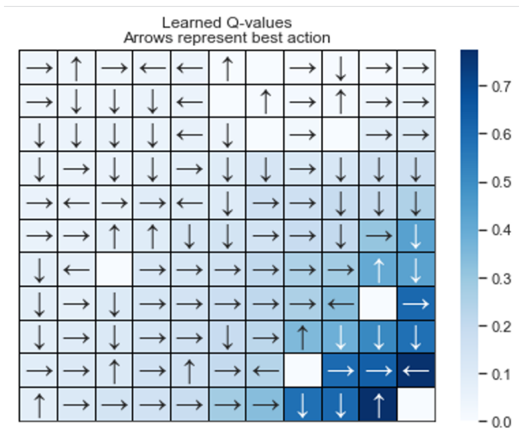


Fig. 4: Policy of Q-Learning

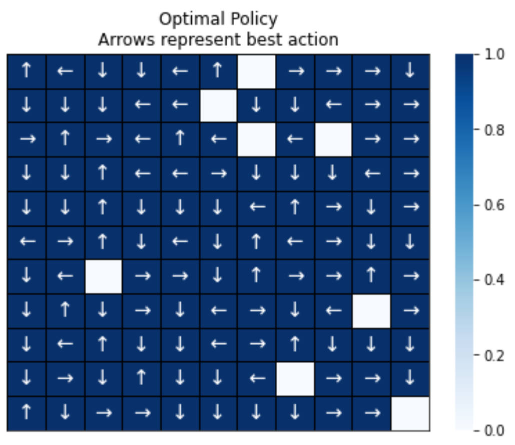


Fig. 5: Policy of MonteCarlo

We see the policy learned by Value Iteration in Fig. 3. We can see that the algorithm has developed what might be the "perfect" policy for completing the game, independent of how many steps does it need to do. Most of the actions are just pointing to the goal, but let's take a look at the tiles near the holes. We can see that the pattern is to move in the opposite direction of the hole, even if it means getting further away of the goal. This is because there is a 66% of chance to slip and don't go to the desired direction, but you can't slip backwards. The map of 11x11 it is very interesting, because we have some holes very close to each other, and we can see some interesting cases like in the third row, where there is a hole on each side horizontally. We can see how the algorithm, decided that it is better to do an action moving to one of the 2 holes, because there is more chance of slipping and surviving, that to actually do the action desired.

In conclusion, Value Iteration has a magnificent performance on frozenlake because all the information needed to get the optimal policy is available since the beginning. Getting a very fast time, and a perfect win rate.

Taking a look at Q-Learning performance, we might think that it has problems to totally understands the challenge in Frozenlake. We have a 53% win rate, which would appear that even though it understands how to complete the game, it has problems dealing with holes.

But in reality, if we see the policy achieved in Fig. 4. Seems that the policy achieved is very similar to Value Iteration. Analyzing the process, we can see that we have a lower win rate because this method needs to do a lot of tries until it really understand the game. Even when we let it play for a large amount of games, it never has more than 60% win rate, that is because Q-Learning doesn't always follow the best action, and tries to explore random actions in order to find new ways of achieving his purpose.

The last algorithm we will see is MonteCarlo, we can observe in Fig. 5 how it is very similar to the others policies. One question we had when looking at it, is how it has a better win rate than Q-Learning when they both work with probabilities and a similar policy. Our answer is that the probabilities of taking an action that isn't the optimal in MonteCarlo are smaller, which makes him win more games.

In summary, we have seen how our algorithms have made different solutions to fronzenlake game, based on how they calculate the optimal policy, but have points in common when they are near a hole.

7 Approximate Solutions

7.1 Models

In this section, we delve into the foundational algorithms shaping our Breakout environment. Consider these algorithms as the sophisticated machinery powering our agent's strategic decisions in the intricate realm of Breakout.

Tabular methods are more accurate but computationally expensive. They rely on discretizing the solution space, which is ideal for simple problems, but on environments like breakout, with a bigger load of states and information, they are too expensive to use. That's why we will use approximate solutions, which are less accurate but faster and more efficient, making them ideal for complex problems.

Let's start with the first one: Deep Q-Learning. DQL is based on our previous algorithm Q-Learning, but now it doesn't save the Q-Values on a table, it uses a neural network instead. The pseudocode is presented in Algorithm 4.

Where:

- γ = Discount factor
- ϵ = Exploration probability
- ϵ_{decay} = Exploration decay rate
- ϵ_{min} = Minimum exploration probability
- α = Learning rate
- B = Batch size
- C = Update frequency

For the second one, we will see Proximal Policy Optimization. It is a more modern algorithm, developed in 2017 by John Schulman, and has very good results in games more complicated like Dota 2, where OpenAi PPO algorithm beat professional players [14]. We won't be deepening much on how it works, due to its complicated structure. We will use it for comparing how good algorithms have been improving these years.

Even though we won't deepen on how it works, we will do a brief pseudocode explication so we can have a basic knowledge about it. The pseudocode is presented in Algorithm 5.

Where:

- γ = Discount factor
- α = Value Updates
- ϵ = Clipping parameter
- K = Number of epochs for optimization
- T = Number of steps per epoch
- B = Batch size
- I = Maximum number of total steps

7.2 Environment

For approximate solutions, we will use Breakout (Fig. 6). Breakout is an Atari game similar to the classical game Pong [15], but is single-player. The main objective is to destroy all bricks. Each time you hit a brick with the ball, it bounces, and the brick is destroyed. If you let the ball fall, you will lose one of the five lives the game gives you.

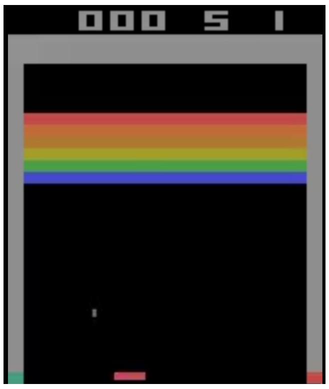


Fig. 6: Breakout screenshot

The challenging part of the game it is that depending on how far from the center of the paddle you hit the ball, the angle and speed of the bounce will be different.

The action space will be a range from 0 to 3. Which will let the player:

0. Do nothing

1. Start the game

2. Move left

3. Move right

The agent gets a reward every time he destroys a block, and the value depends on the color:

- **Red** - 7 points
- **Orange** - 7 points
- **Yellow** - 4 points
- **Green** - 4 points
- **Aqua** - 1 point
- **Blue** - 1 point

Algorithm 4 Deep Q-Learning

```
 1: Initialize replay memory $D$ to capacity $N$
 2: Initialize Q-network with weights $\theta$
 3: Initialize target Q-network with weights $\theta' \leftarrow \theta$
 4: for $t = 1$ to maximum time steps do
 5:     Reset environment and observe initial state $s$
 6:     while $t = 1$ != maximum time steps and not finished (for each episode) do
 7:         With probability $\epsilon$ select a random action $a$, otherwise select $a = \arg\max_{a'} Q(s, a'; \theta)$
 8:         Execute action $a$, observe reward $r$ and new state $s'$
 9:         Store transition $(s, a, r, s')$ in $D$
10:         Sample random mini-batch of transitions $(s_j, a_j, r_j, s'_j)$ from $D$
11:         Compute target values: $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} Q(s'_j, a'; \theta') & \text{otherwise} \end{cases}$
12:         Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ with respect to the network parameters $\theta$
13:         Every $C$ steps, update target Q-network weights: $\theta' \leftarrow \theta$
14:         $\epsilon \leftarrow \max(\epsilon_{\text{min}}, \epsilon_{\text{decay}} \cdot \epsilon)$
15:     end while
16: end for
```

Algorithm 5 Proximal Policy Optimization (PPO)

```
 1: Initialize policy $\pi$ with parameters $\theta$
 2: Initialize value function $V$ with parameters $\phi$
 3: for $step = 1$ to $I$ do
 4:     Collect trajectories using current policy: $\{(s_t, a_t, r_t)\}_{t=1}^{T}$
 5:     for $k = 1$ to $K$ do
 6:         Compute advantages $A_t = \sum_{t'=t}^{T} \left(\gamma^{t'-t} r_{t'} - V(s_{t'})\right)$
 7:         Compute policy loss:
                $$L^{\text{policy}}(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \min\left(r_t(\theta) A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon) A_t\right)$$
 8:         Compute value loss: $L^{\text{value}}(\phi) = \frac{1}{T} \sum_{t=1}^{T} (V(s_t) - (r_t + \gamma V(s_{t+1})))^2$
 9:         Update policy parameters: $\theta \leftarrow \theta - \alpha \nabla_\theta L^{\text{policy}}(\theta)$
10:         Update value function parameters: $\phi \leftarrow \phi - \alpha \nabla_\phi L^{\text{value}}(\phi)$
11:     end for
12: end for
```

The objective is easy, get as many points as you can, and there is no time limit.

Unlike Frozenlake, where all the observation space could be saved on a small structure, breakout complexity and dimensionality is a lot bigger, the ball could be in any pixel of the screen, there are a lot of bricks and different combinations. That's why the observation space in breakout is an actual screenshot of the game, and we can't use Reinforcement Learning algorithms like on last environment. We need Deep Reinforcement Learning algorithms in order to use the screenshots as input.

Before we start training our Deep Reinforcement Learning (DRL) algorithms, we need to optimize our environment. Because the original environment has an observation space of 210 (length) x 160 (width) x 3 (RGB channel) x uint8 (1 byte), which would take a lot of memory usage if we need to do a good training. That's why we need to wrap our environment, and compress it to an 84 (length) x 84 (width) x 1 (grayscale) x uint8 (1 byte), which is 14 times smaller than the original screenshot but doesn't lose any important information.

Compress the screenshot is not the only optimization we will use in our algorithms, in order to make reproduce more episodes in less time, we will stack 4 frames, which means the agent will get their observation, and make a step every 4 frames of the game.

Related to this, but more than an optimization, something necessary. It is that we will stack 4 frames for using as input to our agents. The reason is simple, if we give our agents a screenshot of the game, we lose some valuable information, we can't know where is the ball going, is it up? Down? We can't know at which speed is it going. That's why we will use 4 frames as input to our agent, in order to give him an accurate representation of the movement.

7.3 Results

In this section, we will discuss the results of the algorithms, and understand why they make this performance. The parameters we will use are:

- **Total timesteps = 7000000 (DQL) / 5000000 (PPO)**: We can't limit the time the agent is training by episodes, because it could be in an infinite loop on an episode and never end. So for this environment, we

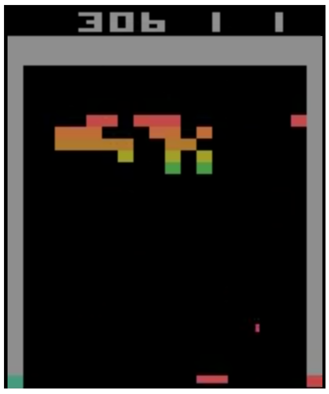


Fig. 7: Screenshot of frame before losing game in DQL

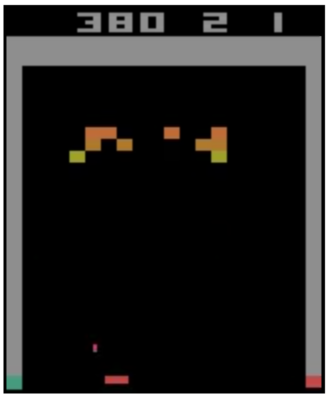


Fig. 8: Screenshot of frame before losing game in PPO

will limit the time training by the maximum number of steps.

- **Buffer size = 900000 (DQL)**: Maximum capacity of the buffer used as memory in DQL.
- **Gradient steps = 50 (DQL)**: Number of steps the neural network will make to update their values in each iteration.
- **Learning rate = 0.00025**: It is the rate at which the model adjusts its weights based on the error.
- **Batch size = 32 (DQL) / 256 (PPO)**: Represents the batch size used for neural network optimization.
- **Gamma = 0.99**: Discount factor.
- **Number of steps per epoch = 2048 (PPO)**: Represents the number of steps in the environment before performing a policy update.

Our first algorithm to compare will be DQL, which had a mean score of 177. We can see in Fig. 7 a screenshot of the last frame of one of their games. We can see how almost all the bricks have been destroyed with a four-hour training. It's interesting to see how the screenshot shows a score of 306, but the mean is way lower. Reproducing more games, we can observe how it is difficult for DQL to react when the game begins or a life is loss, because the ball is lower and has less time to calculate where it is going to fall. For this reason, most of the games have high scores like 300, and low scores like 100, lowering his mean to the value that we get at tensorboard [16]. The training time for our best DQL algorithm has been 10 hours, with seven million timesteps, even though the mean score doesn't have significant improves in the mean score since the fourth million time step, as we can see at section A.2.

Next one is PPO. One thing we observed in the early stages of development, is that PPO is trying to play in a smarter way than DQL. We can see how DQL with low training time is just trying to don't let the ball fall, while we can see how PPO is trying to prioritize the red bricks, which gives the maximum points. Once we got our best parameters, we can observe at table 2 how with less than half the time of training than DQL, PPO has achieved a much higher mean score doubling his opponent with a score of 414. PPO hardly lets the ball fall, but doesn't still understand how to aim for the remaining bricks, to the point that most of the games ends on an infinite loop where the ball never falls, but there are a few bricks that survives, and the game finish after the designated time. We can see in Fig. 8 a screenshot of the last frame of one of their games. As we can observe at section A.2. PPO still has been improving all the training time, and we think that it still can improve more with a bigger training time.

8 Conclusions

We have completed our objectives for this thesis. We have achieved three functional Reinforcement Learning algorithms capable of complete Frozenlake game, and two Deep Reinforcement Learning algorithms capable of complete Breakout game.

We have managed to analyze and understand the behavior of the algorithms and how they have proposed solutions to the challenges we have presented to them. We have enhanced the algorithms by parameterize them based on our capabilities, as we lack sufficient resources to generate automation of optimal parameters. However, with visualization and graphical data representation, we have been able to improve performance.

Learning about Reinforcement Learning using video games as an environment has been very useful to delve into the topic. It has proven to be highly productive, given that the use of these algorithms is valuable in many other areas within the business sector and Artificial Intelligence.

9 Future work

We could have better results with a bigger memory usage, in order to multiprocess the agents and reduce execution time needed. Our hardware and time available has been a big limit in our results, leading to big number of directions for future work:

- It would be worth exploring to analyze one more approximate solution, like *REINFORCE* algorithm.
- Add a new environment more complex than *Breakout* would be interesting to see our approximate solutions limits.
- Improve the hyperparameters with an automated tool like *Optuna* library.
- Create a custom environment

TABLE 2: BREAKOUT ALGORITHMS PERFORMANCE

	Time (seconds)	Mean Score	Memory Usage (MiB)	Energy Usage (CO2 Kg/m^3)
DQL	37305.4	177	11053.53	0.269
PPO	11367.9	415	8713.11	0.065

ACKNOWLEDGMENTS

I am deeply grateful to my thesis tutor Jordi Casas, which has been fundamental for developing this project. Thanks to him, I received a fundamental guidance of what sources and libraries should I research, and provided me with references to use in my code. I would also thank my family for all the moral support that I needed through the development of the project and has been a key part in order to maintain focused in working every day until finishing the project.

REFERENCES

[1] Ruth Brooks, "What is Reinforcement Learning," Last accessed 07 October 2023. [Online]. Available: https://online.york.ac.uk/what-is-reinforcement-learning/

[2] Ph.D.thesis Watkins, C.J.C.H. (1989), "Learning from delayed rewards." Last accessed 07 October 2023. [Online]. Available: https://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf

[3] Google DeepMind, "AlphaGo," Last accessed 07 October 2023. [Online]. Available: https://www.deepmind.com/research/highlighted-research/alphago

[4] The Guardian, "Google's AlphaGo wins second game against Go champion." Last accessed 07 October 2023. [Online]. Available: https://www.theguardian.com/technology/2016/mar/10/google-alphago-ai-wins-second-game-against-go-champion-lee-sedol

[5] Collimator AI, "What is reinforcement learning?." Last accessed 07 October 2023. [Online]. Available: https://www.collimator.ai/reference-guides/what-is-reinforcement-learning

[6] Open AI, "Gymnasium Documentation." Last accessed 07 October 2023. [Online]. Available: https://gymnasium.farama.org

[7] Wikipedia, "Kanban." Last accessed 07 October 2023. [Online]. Available: https://es.wikipedia.org/wiki/Kanban

[8] Asana Inc, "Asana." Last accessed 07 October 2023. [Online]. Available: https://app.asana.com

[9] Richard E. Bellman, "Bellman Equation," Last accessed 20 January 2024. [Online]. Available: https://en.wikipedia.org/wiki/Bellman_equation

[10] Vijay Kanade, "What Is the Markov Decision Process? Definition, Working, and Examples," Last accessed 20 January 2024. [Online]. Available: https://www.spiceworks.com/tech/artificial-intelligence/articles/what-is-markov-decision-process/

[11] Ayoosh Kathuria, "PaperSpace." Last accessed 05 November 2023. [Online]. Available: https://blog.paperspace.com/getting-started-with-openai-gym/

[12] Tim Miller, "Value Iteration," Last accessed 20 January 2024. [Online]. Available: https://gibberblot.github.io/rl-notes/single-agent/value-iteration.html#:~:text=Value%20Iteration%20is%20a%20method,(or%20close%20to%20it).

[13] Anderson, Herbert L., "Metropolis, Monte Carlo, and the MANIAC," Last accessed 20 January 2024. [Online]. Available: https://library.lanl.gov/cgi-bin/getfile?00326886.pdf

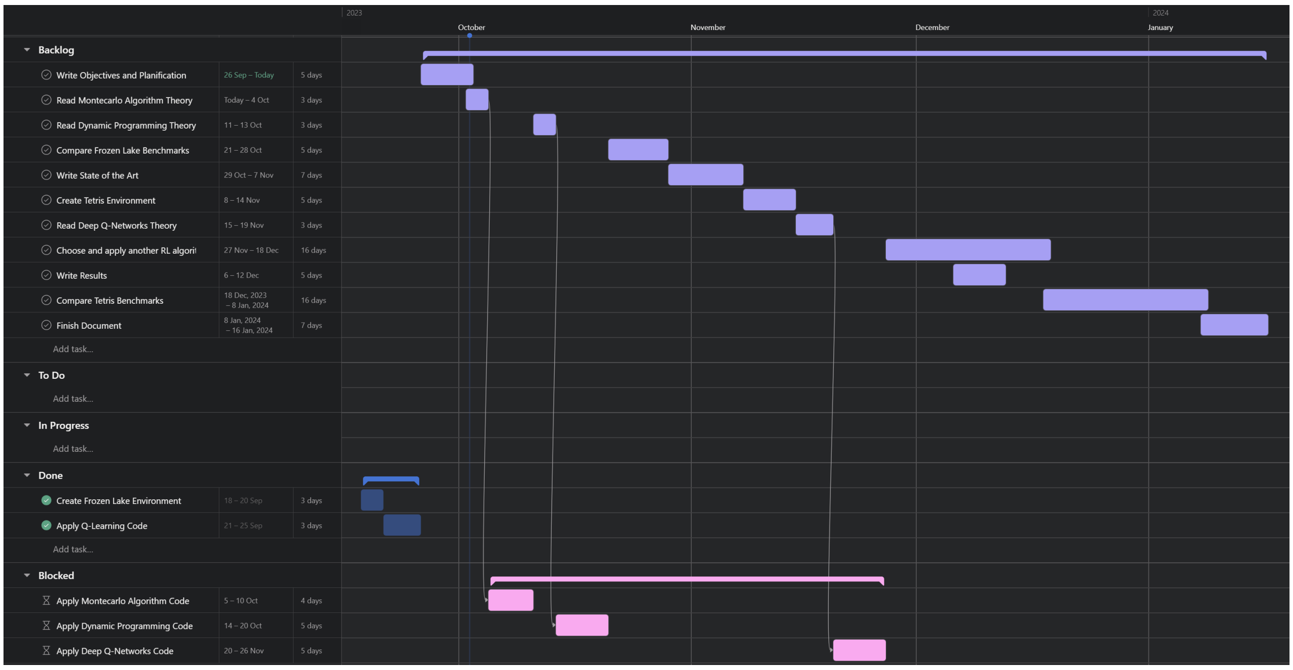
[14] Open AI, "OpenAI Five beats professional players," Last accessed 17 December 2023. [Online]. Available: https://openai.com/research/openai-five-defeats-dota-2-world-champions

[15] Nolan Brushnell, "Pong," Last accessed 16 December 2023. [Online]. Available: https://es.wikipedia.org/wiki/Pong

[16] Tensorflow, "Tensorboard," Last accessed 21 January 2024. [Online]. Available: https://www.tensorflow.org/tensorboard?hl=es-419

APPENDIX

A.1 Planning

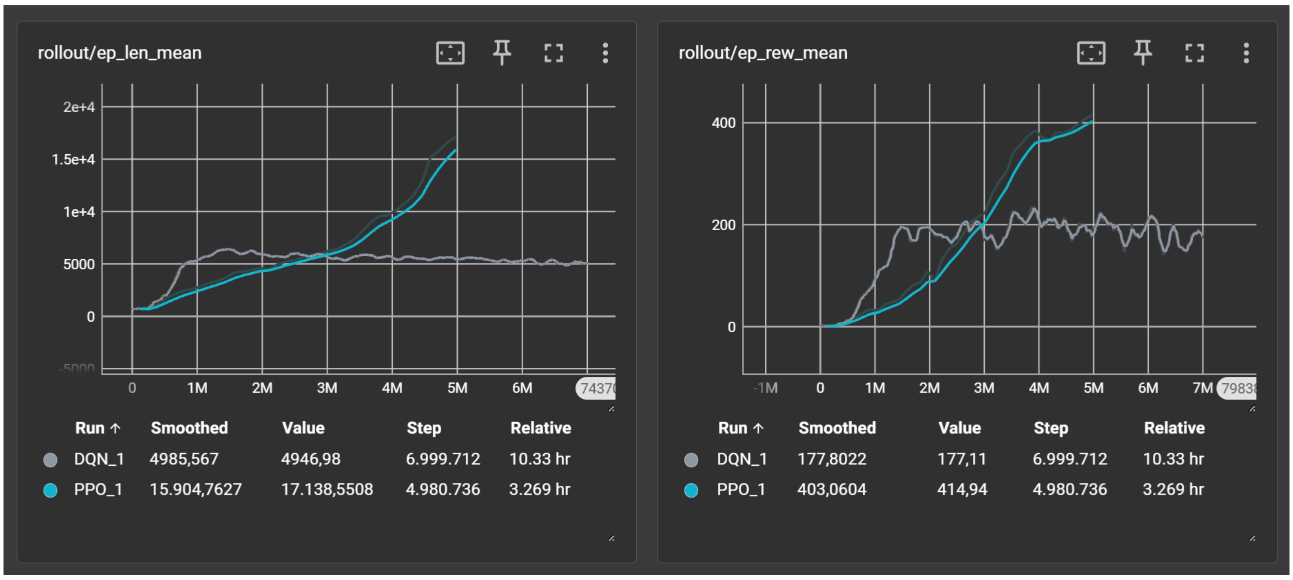


Backlog		
Write Objectives and Planification	26 Sep – Today	5 days
Read Montecarlo Algorithm Theory	Today – 4 Oct	3 days
Read Dynamic Programming Theory	11 – 13 Oct	3 days
Compare Frozen Lake Benchmarks	21 – 28 Oct	5 days
Write State of the Art	29 Oct – 7 Nov	7 days
Create Tetris Environment	8 – 14 Nov	5 days
Read Deep Q-Networks Theory	15 – 19 Nov	3 days
Choose and apply another RL algorit	27 Nov – 18 Dec	16 days
Write Results	6 – 12 Dec	5 days
Compare Tetris Benchmarks	18 Dec, 2023 – 8 Jan, 2024	16 days
Finish Document	8 Jan, 2024 – 16 Jan, 2024	7 days
Add task...		
To Do		
Add task...		
In Progress		
Add task...		
Done		
Create Frozen Lake Environment	18 – 20 Sep	3 days
Apply Q-Learning Code	21 – 25 Sep	3 days
Add task...		
Blocked		
Apply Montecarlo Algorithm Code	5 – 10 Oct	4 days
Apply Dynamic Programming Code	14 – 20 Oct	5 days
Apply Deep Q-Networks Code	20 – 26 Nov	5 days

Fig. 9: Gantt Chart and tasks grouped by their status columns.

Fig. 9 shows the firsts months of the Gantt chart of the project, which is described in more detail in Section 3. This diagram was developed using the objectives as reference for creating and analyzing the needed tasks, including time estimates for each task and their dependencies. The Gantt chart has been an invaluable tool for the project, as it has allowed us to visualize the project's progress and make informed decisions.

A.2 Breakout Benchmarks



rollout/ep_len_mean

Run ↑	Smoothed	Value	Step	Relative
● DQN_1	4985,567	4946,98	6.999.712	10.33 hr
● PPO_1	15.904,7627	17.138,5508	4.980.736	3.269 hr

rollout/ep_rew_mean

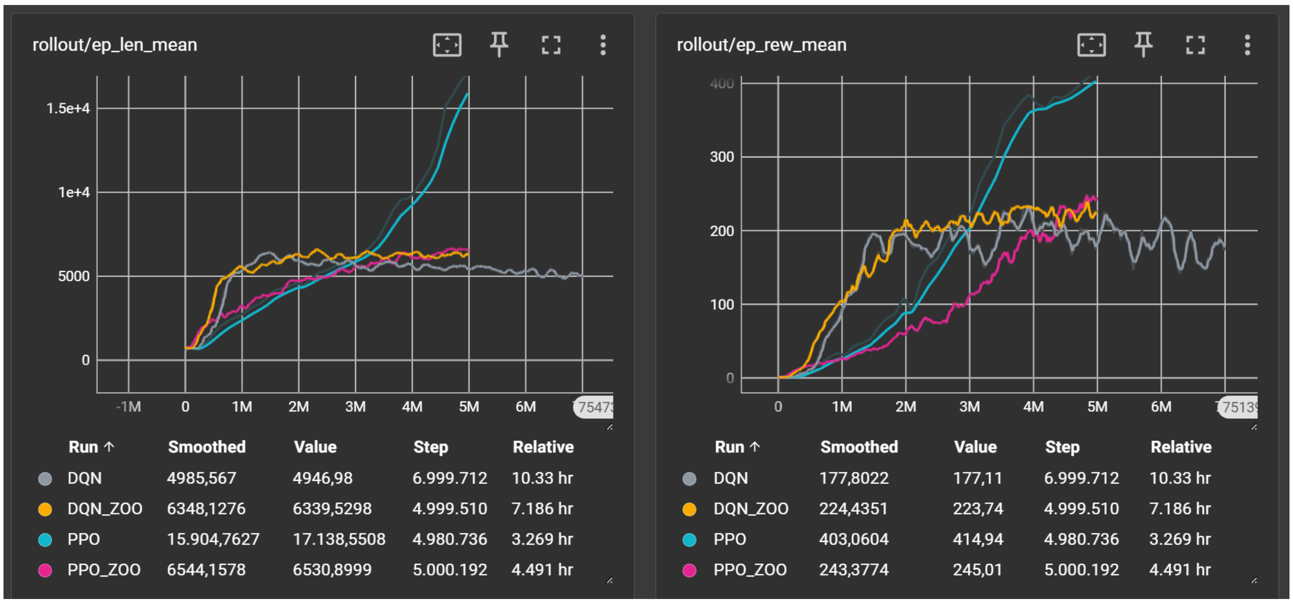
Run ↑	Smoothed	Value	Step	Relative
● DQN_1	177,8022	177,11	6.999.712	10.33 hr
● PPO_1	403,0604	414,94	4.980.736	3.269 hr

Fig. 10: Tensorboard graphic of PPO vs DQL.

Fig. 10 shows the tensorboard graphics of our algorithms PPO and DQL which in this figure has the name *DQN* (Deep-Q Network). *Rollout/ep_len_mean* measures the average length of an episode, which is the number of steps taken before the agent reaches a terminal state. *Rollout/ep_rew_mean* measures the average reward per episode. The graph on the left shows the performance of the two algorithms on length mean. We can observe how the DQN algorithm stagnates at 1 million timesteps, while PPO continues to increase until the end of its execution at 5 million timesteps. Reviewing the videos of the games, we can observe that the PPO time increases so much because it starts to enter a loop in which it does not drop the ball.

The graph on the right shows the performance of the two algorithms on reward mean. As in the previous graph, the DQN algorithm stagnates at 2 million timesteps and its score stops improving, although it fluctuates in an unstable manner, with great variation between each game. The PPO algorithm, on the other hand, continues to improve until 5 million timesteps, where it achieves an average score of 400.

For a better comparison, with the state of the art, we have made a comparison versus the stable baselines 3 zoo library. Stable Baselines 3 Zoo is an open-source Python library that facilitates Reinforcement Learning (RL) development and implementation. It builds on top of Stable Baselines 3 and provides a collection of pre-trained RL agents, training scripts, hyperparameter optimization tools, and other utilities.



rollout/ep_len_mean

Run ↑	Smoothed	Value	Step	Relative
● DQN	4985,567	4946,98	6.999.712	10.33 hr
● DQN_ZOO	6348,1276	6339,5298	4.999.510	7.186 hr
● PPO	15.904,7627	17.138,5508	4.980.736	3.269 hr
● PPO_ZOO	6544,1578	6530,8999	5.000.192	4.491 hr

rollout/ep_rew_mean

Run ↑	Smoothed	Value	Step	Relative
● DQN	177,8022	177,11	6.999.712	10.33 hr
● DQN_ZOO	224,4351	223,74	4.999.510	7.186 hr
● PPO	403,0604	414,94	4.980.736	3.269 hr
● PPO_ZOO	243,3774	245,01	5.000.192	4.491 hr

Fig. 11: Tensorboard graphic of PPO vs DQL.

It is important to note that the analyzed zoo algorithms are incomplete. We have limited them to the same number of timesteps as ours to make a better comparison. However, the zoo library algorithms have many more timesteps and can reach a score of 500, making a perfect game. We can observe at Fig. 11 that our DQN algorithm has a similar behavior to the DQN Zoo algorithm. In contrast, it can be seen that our PPO algorithm has achieved a much greater advantage in the short training time, almost doubling the average score.

In conclusion, our results are very good for the objective we had, which was to achieve the maximum possible score with a shorter training time. However, we cannot assure if any of our parameter adjustments are better than the current ones in the zoo library, since we do not have enough execution time to see if our algorithms can reach the maximum score.