Reinforcement Learning in Videogames

David Fuentes Insa

15 de juny de 2025

Resum– Aquest projecte té com a objectiu establir una base sòlida sobre l'aprenentatge per reforç, entenent-ne els fonaments, aprenent a implementar-lo a través de diversos algoritmes i explorant com pot aplicar-se als videojocs. Amb l'augment de la importància de la intel·ligència artificial en múltiples sectors, l'aprenentatge per reforç destaca com un paradigma potent per entrenar agents a prendre decisions mitjançant la interacció i l'experiència. L'estudi comença amb una anàlisi de conceptes teòrics com els processos de decisió de Markov, els entorns, les recompenses, les polítiques i els agents. S'estudien, implementen i avaluen algoritmes de RL tabulars i no tabulars aplicats a agents individuals en entorns de videojocs. Els videojocs s'utilitzen com a entorns interactius i controlats per observar el comportament dels agents i el rendiment dels algoritmes. Finalment, l'estudi aprofundeix en tècniques d'entrenament multiagent, significativament més complexes que l'entrenament d'un sol agent.

Paraules clau– Aprenentatge per reforç, intel·ligència artificial, aprenentatge d'un sol agent, aprenentatge multiagent, videojocs, aprenentatge profund

Abstract– This project aims to get a solid base of Reinforcement Learning, understanding the foundations, learning how to implement it across multiple algorithms and how can be applied to videogames. As AI becomes increasingly important across many industries, RL stands out as a powerful paradigm for training agents to make decisions through interaction and experience. The study begins with an analysis on theoretical concepts such as Marko Devision Processes, enviroments, rewards, policies and agents. Tabular and non-tabular RL single agent algorithms are studied, implemented and evaluated, in videogames enviroments. Videogames are used as controlled, interactive enviroments to test learning behavior and algorithm performance. The study then proceeds to dive into multi-agent training techniques, much more complex than training a single-agent.

Keywords– Reinforcement Learning, Artificial Intelligence, Single-agent learning, Multi-agent learning, Videogames, Deep learning

1 Introduction

Reinforcement learning is one of the most promising approaches of artificial intelligence, especially in the videogames sector. Meanwhile other AI approaches use supervised training with labeled data, RL agents learn by interacting with the enviroment throught trial and error, and improve based on rewards or penalties for taking actions.

The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them. Inthe most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics —trial-and-error search and delayed reward— are the two most important distinguishing features of reinforcement learning [1].

In 2013, the company DeepMind released a paper regarding how their RL models could surpass a human expert playing Atari 2600 classic games. The model beat the human expert in 3 of 6 games tested [3]. In 2015, they released a more extended paper tested on 49 games. The model got to the level of a human professional [4]. The most impressive archievement is AlphaGo, the model who beat the European Go champion Fan Hui by 5 games to 0 [5].

• E-mail de contacte: david.fuentesinsa@gmail.com
• Menció realitzada: Computació
• Treball tutoritzat per: Jordi Casas Roma
• Curs 2024/25

Videogames provide an ideal enviroment to test the models because of the clear objectives and controlled actions. However, reinforcement learning applications extend not only in videogames, also in real life problems such as robotics, autonomous vehicles, healthcare, resource management, and much more.

2 OBJECTIVES

The main objective of this project is fully understanding Single-agent Reinforcement Learning and what can be acomplished with it using existing libraries but also how it works from zero. The next five objectives define the goal by the end of the project.

1. Learn RL Fundamentals: Study core RL concepts such as Markov Decision Processes (MDPs), value functions, and policy optimization. Implement simple algorithms like Monte Carlo or Dynamic Programming to build a solid base.
2. Learn RL Advanced Methods: Such as Temporal Difference learning (Q-learning and SARSA) and deep RL approaches like DQNs.
3. Implementing some algorithms from zero, without using pre-built libraries.
4. Implementing image based DQNs that are more complex and harder to train.
5. Experiment with self-play, multi-agent training in a videogame: Using a ping pong enviroment, train the agents from both sides of the game.

3 METHODOLOGY AND PLANNING

The methodology chosen is Kanban, an agile approach that has a clear visualization of the workflow, the evolution is continuous and is very flexible.

The tool used for the implementation of Kanban is Clickup [2], a free tool for project management.

The planning is divided in five phases of three or four weeks each.

- Phase 1: Reinforcement learning fundamentals and the State of Art (10/03/25 - 19/04/25)
  - Task 1: Research and document core RL concepts (agents, environments, states, actions, rewards)
  - Task 2: Study tabular methods (Dynamic Programming, Monte Carlo, Q-learning, SARSA)
  - Task 3: Study non-tabular methods (DQN)
  - Task 4: Document the State of Art
- Phase 2: Enviroment setup and algorithms (18/04/25 - 25/05/25)
  - Task 5: Study gymnasium library
  - Task 6: Implement non-tabular methods
  - Task 7: Implement tabular methods
  - Task 8: Document the implementations and tests
- Phase 3: Deep Q-Network (DQN) applied to image-based game (04/05/25 - 25-05-25)
  - Task 9: Select enviroment and study the DQN algorithm
  - Task 10: Implement and train the agent
  - Task 11: Test the algorithm
  - Task 12: Document the implementations and tests
- Phase 4: Self-play (2025-05-20 - 2025-06-14)
  - Task 13: Study petting zoo library
  - Task 14: Implement the training and testing algorithm
  - Task 15: Train the agents implemented
  - Task 16: Test the model, and evaluate the models performance
  - Task 17: Document the implementations and tests
- Phase 5: Final inform and conclusions
  - Task 18: Document the and results conclusions
  - Task 19: Final inform
  - Task 20: Project presentation

4 STATE OF ART

4.1 Reinforcement Learning Fundamentals

These are the basic components a reinforcement learning problem has.

An agent — it's the entity that makes decisions. It's objective is to learn a policy ($\pi(a|s)$) that maximizes the accumulated reward. It can be a robot, a player in a videogame, etc. Enviroment — responds to the agent actions. It provides the states and the rewards, and changes based on the agent actions. States — Represent the actual enviroment situation by the perspective of the agent. Actions — The possible decissions an agent can do in a determinate state s. Choosing an action makes a transition to a new state. Rewards — Numeric values that indicate the immediate utility of an action. The agent must maximize the rewards at long term. An immediate reward r_t is a reward obtained in the time t. The return G_t, is the future rewards sum with a discount factor γ, which determines how much the agent values future rewards. If its near 0, the agent only cares for immediate rewards. If it's near 1, the agent will appreciate much more the future rewards.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Policty — Defines the agent behavior. It's the strategy the agent follows. $\pi(a|s) = P(A_t = a \mid S_t = s)$

What would these components be in a reinforcement learning problem applied in the videogame of Space Invaders?

The agent, is the spaceship that moves horizontally and shoots the aliens. The enviroment, would be the aliens, the visual scenary, the alien bullets and all the game interface like lives, points. The state could include: position of the agent spaceship, position of all the aliens, position of every bullet, the lives remaining and the image frame. The actions: move left, move right, shoot, doing nothing. The rewads: +1 for shooting an enemy, -1 for loosing a live and 0 for not doing anything for example. The politic: If an enemy is just up the agent, shoot.

4.2 Markov Decision Process (MDP)

A Markov Decision Process is the math model that formalizes a RL problem. It is defined by a tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$ where $\mathcal{S}$ is the set of all possible states, $\mathcal{A}$ is the set of all possible actions, $P(s'|s,a)$ is the probability of transitioning to state s' when taking action a in state s, $R(s,a)$ is the expected immediate reward received after taking action a in state s, and $\gamma \in [0,1]$ is the discount factor.

The Markov property states that the probability of transitioning to the next state depends only and only on the current state and action, not on the sequence of previous states and actions:

$$P(s_{t+1}|s_t, a_t) = P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \dots)$$

4.3 Tabular methods

Ideal when the enviroment is small and manageable. State-action values can be stored in tables.

4.3.1 Dynamic Programming

Based on a simple idea, if we know exactly how the enviroment works (what will happen with a probability when we take an action in a specific state), we can calculate which decisions are better in long term. This is very limited, because in the real world we don't know complete knowledge of the enviroment [6].

There are three algorithms used in DP. Policy evaluation: Calculates how good is the current policy using Bellman equation in every state using that specific policy.Policy iteration: Firstly uses policy evaluation and then improves the current policy choosing the best actions based on the calculated values.Values iteration: This method updates the values of every state with a simplified Bellman equation, to get the optimal policy.

4.3.2 Monte Carlo

Does not need to know the model of the enviroment. Learns based on experience.

The monte carlo algorithms works like this: The agent follows a politic [7]. Plays an episode till the end. At the end of the episode, calculates how much has won in total from every state. Updates the value estimates. Improves the politic, to take better actions in the next episodes

What is an episode? A sequence of states starting from one state to the end state when a condition is met.

There are two main objectives in Monte Carlo algorithms. Prediction: Follow a specified policy and evaluate how good the policy is. Control: To learn the best actions the agent can do, keep adjusting the policy.

And there are two approaches on how to update what the agent learnt. First-visit: Updates the value of the state only the first time it appears on an episode. Every-visit: Updates the value of state every time it appears

4.3.3 Temporal Difference

Learn from previous experience and don't need to end an episode like Monte Carlo algorithms. They are ideal for tasks that do not have an end (like maintaining the temperature of an oven that is always on, or to move a character in a videogame on an open world without a clear objective). Combines ideas from Monte Carlo and Dynamic Programming [8].

TD algorithms update the value of a state based on future estimates. When the agent changes the state and gets a reward, the state value gets immediately updated. This update is based on the **temporal difference (TD) error**, which measures the difference between the current Q-value and a new estimate based on the reward received and the next state's predicted value. It is the core learning signal used to adjust the Q-value closer to reality.

There are two main algorithms based on TD: SARSA — means State-Action-Reward-State-Action. To learn, must know the state, the chosen action, the reward obtained, the next state and the next chosen action. Learns from what you are doing, even if that is not the best possible.

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma Q(s',a') - Q(s,a)\right]$$

Q-learning — is almost the same, but learns from the best action possible.

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$$

4.4 Non-Tabular methods

Methods where every action-state value cannot be stored on a table because it will be infinite or impossible to manage.

4.4.1 DQN

Deep Q-Network (DQN) combines Q-learning with deep neural networks [4]. The q table is replaced for a neural network, that can estimate the q values.

How do DQN Networks work? You send the actual state to the network input (for example, the game image or the needed variables) and the network outputs the estimate q-values for every action possible. Then, the greatest q-value is chosen and executed, and get the new state and the reward. It utilizes temporal difference to calculate the error between the predicted Q-value and a new estimate based on the reward and the next state. Finally, the DQN Network is updated.

4.4.2 REINFORCE

Reinforce [11] is a policy gradient algorithm that learns how to act directly, instead of learning how good each action is (like DQN methods). Instead of using a Q-table or estimating values, Reinforce tries to learn the best policy, the best

way of acting, by adjusting the probabilities of choosing an action in each situation.

How do Reinforce works? It plays an episode, then it looks what happened. If an action led to a good reward, it will make that action more likely to be chosen next time. Otherwise, if an action led to bad rewards, it will make it less likely to be chosen. Gradually improves how it plays.

5 Single-Agent Learning

5.1 The gymnasium library

Gymnasium [13] is a python library, forked from OpenAI Gym Library, that provides a standarized interface for creating enviroments where the agents can interact and learn from. The library provides a clear API aligned with the Markov Decision Processes where the RL agents can observe a state, choose an action and receive a reward.

CartPole-v1 and FrozenLake-v1 are two Gymnasium enviroments that are used in the tabular and non-tabular implementations of RL algorithms.

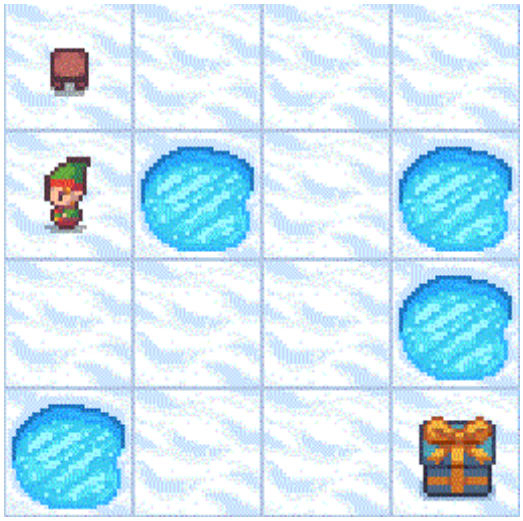

Fig. 1: FrozenLake-v1: A discrete grid. The agent must learn to navigate a frozen lake without falling into holes.

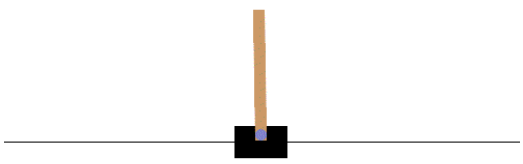

Fig. 2: CartPole-v1: A classic control problem with continuous states. The goal is to balance a pole on a moving cart.

5.2 Implementing tabular methods

5.2.1 Implementation of the algorithms

Three tabular methods were implemented and tested on the FrozenLake-v1 enviroment: Monte Carlo, SARSA and Q-Learning. FrozenLake has an important enviroment option: slippery. If slippery is true the game is not deterministic and the agent has a 30% chance to slide and choose a random The agent can choose from 4 discrete actions: left, right, up, down. Reward shaping was implemented because the original FrozenLake enviroment provides plain and binary rewards: 1.0 reward if the agent reaches the goal, 0.0 otherwise. To accelerate learning and guide the agent more effectively, the following shaped rewards were applied: 1.0 when reaching the goal, -1.0 if the agent falls into a hole, -0.01 if the agent does a step.

Algorithm 1 Monte Carlo Control (First-Visit)

Initialize $Q(s,a)$ arbitrarily
for each episode **do**
  Initialize empty list $\mathcal{E} \leftarrow []$ ▷ stores (s,a,r)
  Initialize state s
  while episode not ended **do**
    Choose action a using ε-greedy policy
    Take action a, observe reward r and next state s'
    Append (s,a,r) to $\mathcal{E}$
    $s \leftarrow s'$
  end while
  for each (s,a) in $\mathcal{E}$ (first visit only) **do**
    Compute $G \leftarrow$ total discounted return after first occurrence
    $Q(s,a) \leftarrow Q(s,a) + \alpha \cdot [G - Q(s,a)]$
  end for
end for

Algorithm 2 SARSA (On-policy TD Control)

Initialize $Q(s,a)$ arbitrarily
for each episode **do**
  Initialize state s
  Choose action a using ε-greedy policy
  while episode not ended **do**
    Take action a, observe reward r and next state s'
    Choose next action a' using ε-greedy policy
    $Q(s,a) \leftarrow Q(s,a) + \alpha \cdot [r + \gamma Q(s',a') - Q(s,a)]$
    $s \leftarrow s'$
    $a \leftarrow a'$
  end while
end for

Algorithm 3 Q-Learning (Off-policy TD Control)

Initialize $Q(s,a)$ arbitrarily
for each episode **do**
  Initialize state s
  while episode not ended **do**
    Choose action a using ε-greedy policy
    Take action a, observe reward r and next state s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha \cdot [r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
    $s \leftarrow s'$
  end while
end for

5.2.2 Results and Conclusions from Training and Evaluation

All the same hyperparameters were used for every algorithm: 100000 episodes, 150 max steps of an episode, alpha (α) 0.1, gamma (γ) 0.99, 1.0 initial epsilon, 0.9999 epsilon decay every episode, and the minimum epsilon 0.01. The enviroment was set to 8x8 grid with 0.8 probability of holes. 30000 episodes and 150 max steps were used for testing every algorithm.

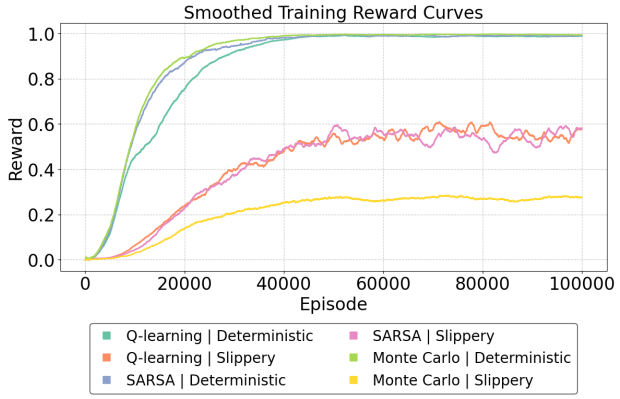

Fig. 3: Smoothed training reward curves

All the algorithms performed with good results in the deterministic enviroment archieving a 100% success on all the tests. Monte Carlo was a little bit slower in the training of deterministic enviroment and performed worse when using non-deterministic enviroment. Q-learning performed the best on non-deterministic archieving more than a 70% success. Monte Carlo performed poorly on the non-deterministic reaching just less than 35% success rate. The slippery option is kind of experimental and depending on the map generated, it may be impossible the model to not fall in a hole just because of randomness.

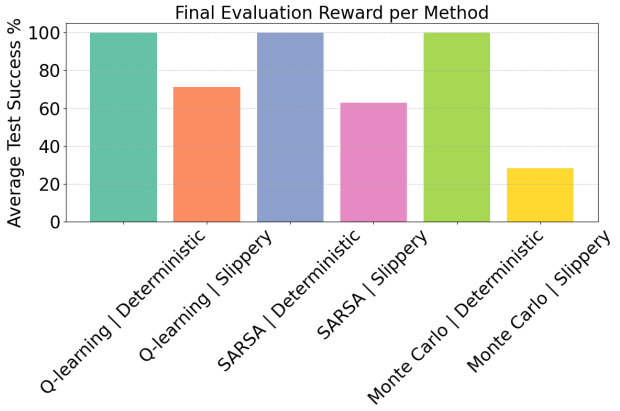

Fig. 4: Tested success (%)

5.3 Implementing non-tabular methods

5.3.1 Implementation of the algorithms

One non-tabular method was implemented on the CartPole-v1 enviroment: Deep Q-Network. The Q-Network consists of two hidden layers with 64 neurons each and ReLU activations. The input of the network is the observable space of the enviroment: Cart position, cart velocity, pole angle and pole angular velocity. The output is 2 discrete actions: moving the cart left and moving the cart right.

Algorithm 4 Deep Q-Network (DQN)

Initialize Q-network with random weights
Initialize replay buffer $\mathcal{B}$
for each episode **do**
  Initialize state s
  for each step of the episode **do**
    if random number $< \varepsilon$ **then**
      Choose random action a
    else
      $a \leftarrow \arg\max_a Q(s, a)$
    end if
    Execute action a, observe reward r and next state s'
    Store $(s, a, r, s', \mathit{done})$ in buffer $\mathcal{B}$
    $s \leftarrow s'$
    if buffer $\mathcal{B}$ has enough samples **then**
      Sample minibatch of transitions $(s, a, r, s', \mathit{done})$ from $\mathcal{B}$
      for each transition **do**
        if *done* **then**
          target $\leftarrow r$
        else
          target $\leftarrow r + \gamma \cdot \max_{a'} Q(s', a')$
        end if
        Compute loss: $\mathcal{L} = (Q(s, a) - \text{target})^2$
        Update Q-network to minimize $\mathcal{L}$
      end for
    end if
  end for
  $\varepsilon \leftarrow \max(\varepsilon_{\min}, \varepsilon \cdot \text{decay\_rate})$
end for

5.3.2 Results and Conclusions from Training and Evaluation

The hyperparameters were: 1000 episodes, 200 test episodes, 500 max steps of an episode in training and 1000 in testing, 32 batch size, gamma (γ) 0.99, alpha (α) 1e-4, 1.0 initial epsilon, 0.99 epsilon decay every episode and minimum epsilon 0.1.

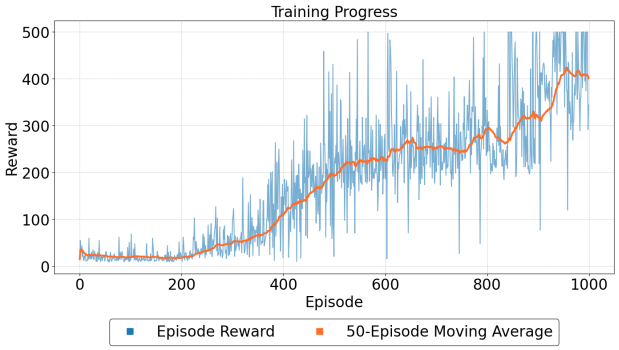

Fig. 5: Smoothed training reward curves

200 episodes and 1000 max steps were used for testing the algorithm. The best model using a 10 episode average was saved and used for the tests. The model performed very good even with very little episodes in the training. The agent is performing more than 500 steps even reaching 1000

sometimes. Maybe if the training was done with more than 500 max steps could get a better result.

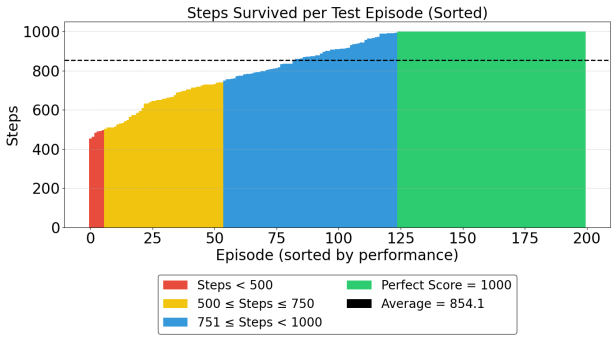

Fig. 6: Steps archieved for every test episode

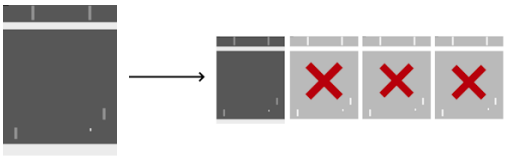

Fig. 8: Frame skipping, getting a frame every 4

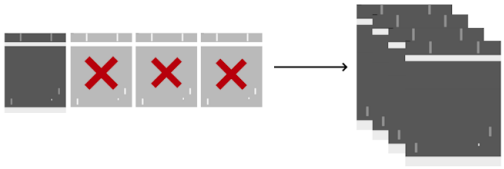

Fig. 9: Frame stacking, getting 4 frames for the input

5.4 Deep Q-Network applied to image based game

5.4.1 Implementation of the algorithms

Pong from atari is a classic Atari 2600 game and it's the enviroment it will be used to traing and evaluate a reinforcement learning dqn. The difference between this enviroment and the CartPole-v1, which we also trained with a DQN, is that we now have a bigger input in the deep neural network. The input is the preprocessed [Fig. 7] 84x84 in greyscale images. Also FrameSkip [Fig. 8] of 4 is used (4 frames) to remove useless in-between frames. Then FrameStack [Fig. 9] of 4 is used (4 frames), to capture motion. A deep neural network [Fig. 10] with 3 convolutional layers (extracting spatial and motion features), followed by a flattening layer and 2 fully connected layers, ending in 6 outputs representing the Q-values for each possible action. In the training, the agent utilizes a epsilon-greedy policy to explore the space of actions. The model was trained on the cloud using Kaggle [17] using GPU acceleation for faster iteration cycles and better performance. To stabilize training, the DQN architecture uses two separate networks: the policy network and the target network. The policy network is the one being actively trained and used to select actions, while the target network is a copy of the policy network that is updated less frequently (e.g., every few thousand steps) and is used to compute the target Q-values. This separation helps reduce oscillations and divergence during training.

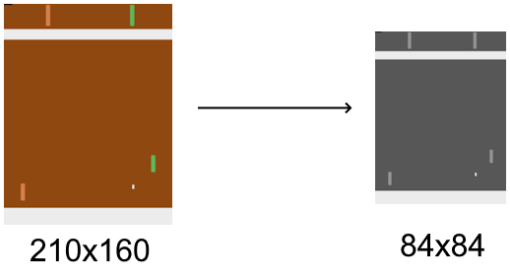

Fig. 7: Preprocessing

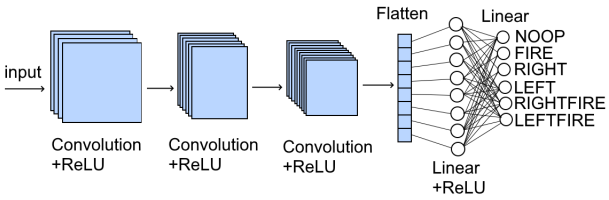

Fig. 10: Deep Q Network, input is 4 frames stacked and ouput the best actio

5.4.2 Results and Conclusions from Training and Evaluation

The training hyperparamters: 8M frames (11200 episodes), gamma (γ) 0.99, alpha (α) 1e-4, Batch-size 32, buffer size 100k, buffer started with 50000 random actions. In the training figure [Fig 11], epsilon started with 1 to 0.1 from A to B. Great progress can be seen here – and this is normal, because the agent is doing a lot of exploration (random actions), which makes the model take non correct actions. When the epsilon decays, the model improves proportionally. From B to C the epsilon was static in 0.1. A little progress was made here, maybe if more frames were trained with the epsilon on 0.1, the model would have get better. A slight progress is seen. And finally, from C to the end epsilon was changed to 0.0 to test if that could give the model a bit more progress. No progress or very little progress is seen.

Some limitations the training process had – 32 GB of RAM which limited the buffer size to 100k. This contrasts with the original 2015 DQN implementation by Mnih et al. [4], which used a replay memory buffer of 1 million transitions, allowing the agent to sample a much wider variety of past experiences during training and improving the stability and performance of learning.

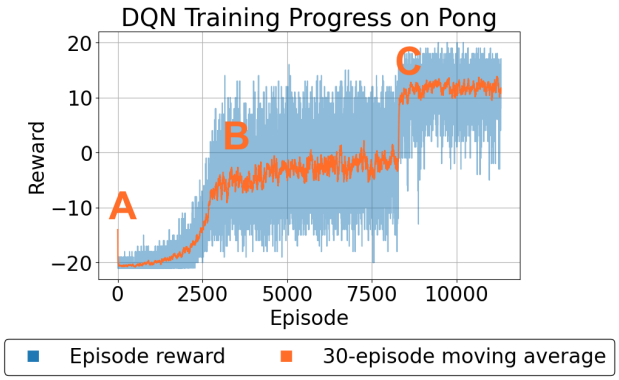

Fig. 11: Smoothed training progress

To choose the best model possible to evaluate, 3 options were tried on 200 episodes. The best average reward model in training within 10 episodes window, the last model when training finished and the best reward episode (an episode reaching 20 reward). The model that had the best performance was the best average reward model in training with 12.61 average reward, is the model in the figures [Fig. 12] and [Fig. 17]. The other two models had 12.35 (last model after training) and 11.14 (the best episode model, with reward of 20). The model performed very well winning 199 out of 200 test episodes. Mostly, the model wins by more than 7 points and more than half of the time it wins by more than 13 points. The model is very robust.

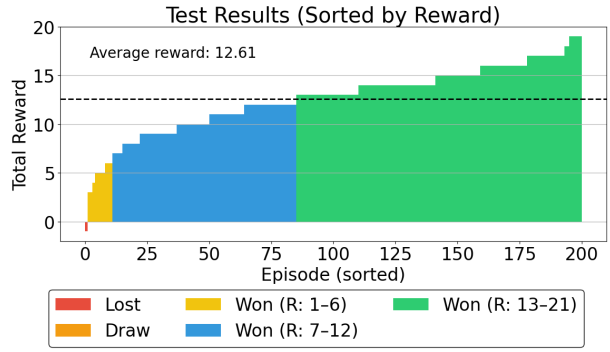

Fig. 12: Testing results, sorted

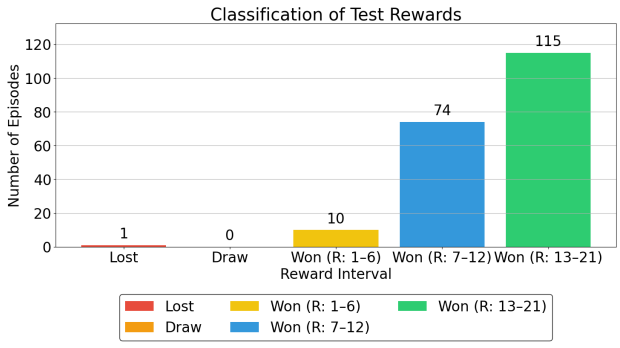

Fig. 13: Testing results, by reward intervals

6 Multi-Agent Learning (MARL)

In Multi-Agent Reinforcement Learning (MARL), multiple agents learn within a shared enviroment. Unlike traditional single-agent RL where the enviroment is stationary, MARL introduces complexity due to the enviroment being non-stationary. An enviroment is stationary when the transition probabilities and the reward function do not change over time. MARL is non-stationary because the other agents will change their strategies while training, so the current model strategy that works now, may not work in the future when the other/s agents changes his strategy. The agents can be competitive, cooperative or both.

6.1 The petting zoo library

PettingZoo [14] is a Python library designed to standarize enviroments specific for MARL learning. Like Gymnasium [13], it provides a unified API and a diverse set of enviroments for testing and benchmarking MARL algorithms. Can only be run in Linux or macOS.

6.2 Implementing the MARL algorithm. DQN vs REINFORCE

The enviroment used is PettingZoo pong [15]. The agents will be competitive. Unlike the environment used in Gymnasium's [13] Pong implementation, in PettingZoo's Pong environment, if the agent does not serve the ball, it gets penalized: "Serves are timed: If the player does not serve within 2 seconds of receiving the ball, they receive -1 points, and the timer resets. This prevents one player from indefinitely stalling the game, but also means it is no longer a purely zero-sum game"[15]. Now instead of implementing my own preprocessing, frame stacking and frame skipping methods, im using pre-built ones from python library supersuit [16]. Using frameskipping and framestacking of 4 like the Single-agent pong. To stabilize the training from both agents, starting on 1M frames and beyond, every 50k frames the training for one of the agents is freezed and the other one enabled. The DQN model is the same implementation as the single agent but using petting zoo enviroment and supersuit wrappers. The REINFORCE agent utilizes a neural network with a softmax final layer to generate an action probability distribution. It chooses actions randomly for the first 75,000 steps to promote early exploration. Afterwards, it samples actions from the policy and accumulates log-probabilities and their respective rewards. At the end of an episode, it rewards actions leading to better results and adjusts the network so it is more likely to make such actions again. It also adds a small amount of randomness (entropy) so it can continue exploring and prevent repeating the same actions. This time the model is trained in a local PC using GPU 5070.

Algorithm 5 Reinforce algorithm

Initialize policy network with random weights
Initialize empty lists for log-probabilities and rewards
for each episode **do**
    Reset the environment and get initial state s
    while episode not finished **do**
        if still in pretraining (e.g., less than 75000 frames) **then**
            Choose a random action a
        else
            Convert state to tensor and pass through policy network
            Get action probabilities and sample an action a
            Save $\log(\text{probability of } a)$ to log-prob list
        end if
        Take action a, get reward r and next state s'
        Add r to reward list
        $s \leftarrow s'$
    end while
    Calculate discounted returns G_t from rewards
    Normalize G_t (mean 0, std 1) and clip between $[-2, 2]$
    Compute loss: $-\log(\text{prob}) \times G_t$ for each step
    Add entropy bonus to encourage exploration
    Update policy network using optimizer
    Clear log-probabilities and rewards for next episode
end for

6.2.1 Results and Conclusions from Training and Evaluation

TODO

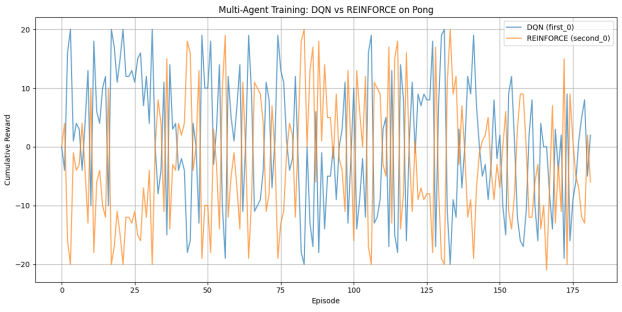

Fig. 14: Testing results, by reward intervals

To evaluate, two distinct methods will be used. The first, to see which agent is better, the episode reward is compared. The second, to see if the models are well trained and competitive, the episode lenght in frames will be compared to episode lenght of random action models.

Trained with 2M frames, and 200 test episodes.

TODO

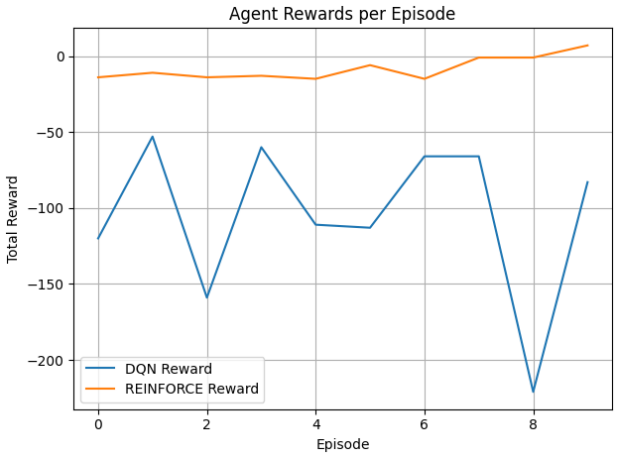

Fig. 15: Testing results, by reward intervals

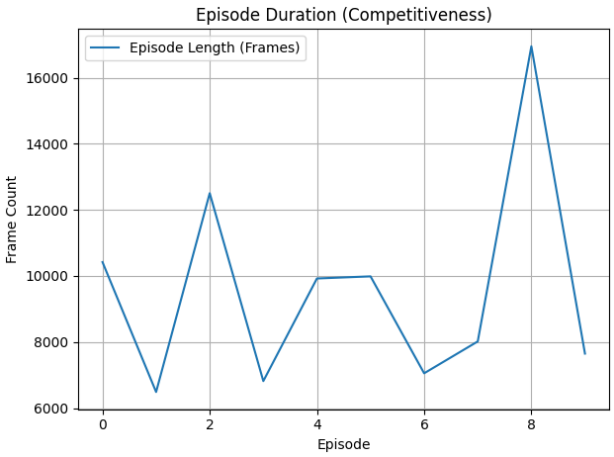

Fig. 16: Testing results, by reward intervals

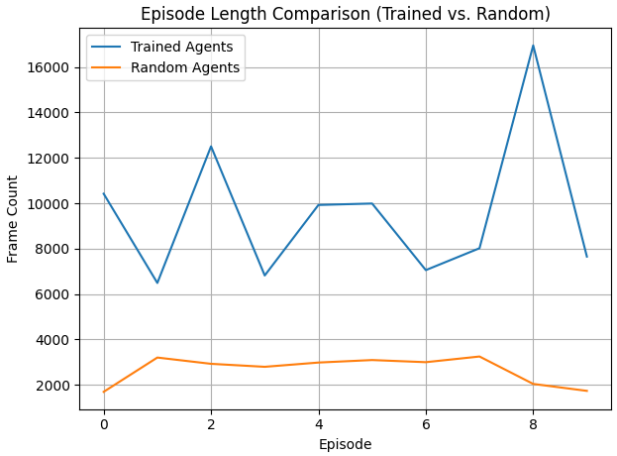

Fig. 17: Testing results, by reward intervals

7 Results and Final Conclusions

All the implementations led to robust results. In the non-tabular methods every model could beat the FrozenLake-v1 game. The tabular implementation of the DQN on the CartPole-v1 was also pretty impressive getting to 1000 steps just having the model trained with 500 max steps and

with so few episodes. The image based DQN implementation was successfull too, because it almost always won the machine (199 wins out of 200 matches).

Hyperparameter complexity is one of the biggest difficulties when it comes to working with reinforcement learning (RL). Factors such as learning rate, discount factor, and exploration parameters can have a massive influence on whether an agent succeeds or completely fails. It gets worse still due to the behavior of each environment being different. Something that works in a given game or simulation may not in another, so each experiment tends to need manual tweaking and trial and error. Cost of training is another key consideration. Having to train a model for hours and not getting the expected result can be frustrating sometimes, and costing a lot of hours trying to get your desired performance on the model.

Referències

[1] Sutton & Barto (2018). Reinforcement learning: an introduction.

[2] https://clickup.com/about

[3] Mnih, Volodymyr and Kavukcuoglu, Koray and Silver, David and Graves, Alex and Antonoglou, Ioannis and Wierstra, Daan and Riedmiller, Martin. Playing Atari with Deep Reinforcement Learning. (2013).

[4] Mnih, V., Kavukcuoglu, K., Silver, D. et al. Human-level control through deep reinforcement learning. Nature 518, 529–533 (2015).

[5] Silver, D., Huang, A., Maddison, C. et al. Mastering the game of Go with deep neural networks and tree search. Nature 529, 484–489 (2016).

[6] Bellman, R. (1957). A Markovian decision process. Journal of Mathematics and Mechanics, 6(5), 679–684.

[7] Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. IBM Journal of Research and Development, 3(3), 210–229.

[8] Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. Machine Learning, 3, 9–44.

[9] Rummery, G. A., & Niranjan, M. (1994). On-line Q-learning using connectionist systems (Tech. Rep. No. CUED/F-INFENG/TR 166). Cambridge University Engineering Department.

[10] Watkins, C. J. C. H. (1989). Learning from Delayed Rewards (PhD thesis). University of Cambridge.

[11] Williams, Ronald J. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. (1992).

[12] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). OpenAI Gym. *arXiv preprint* arXiv:1606.01540.

[13] Farama Foundation. (2022). Gymnasium: A fork of OpenAI Gym. https://github.com/Farama-Foundation/Gymnasium

[14] https://pettingzoo.farama.org/

[15] https://pettingzoo.farama.org/environments/atari/pong/

[16] https://github.com/Farama-Foundation/SuperSuit

[17] Kaggle.com https://www.kaggle.com/

8 Appendix

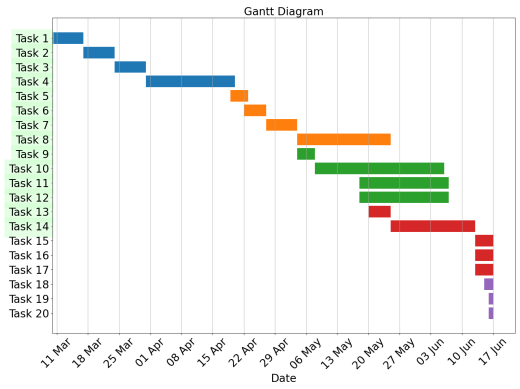

Fig. 18: Grantt diagram

Project github: https://github.com/deivvvid/TFG-RL (Still not pushed)