

# Reinforcement Learning in Videogames

David Fuentes Insa

25 de maig de 2025

**Resum**– Resum del projecte, màxim 10 línies. ....  
 ....  
 ....  
 ....  
 ....  
 ....  
 ....  
 ....  
 ....  
 ....  
 ....

**Paraules clau**– Paraules clau del treball, màxim 2 línies . ....  
 ....

**Abstract**– Versió en anglès del resum . ....  
 ....  
 ....  
 ....  
 ....  
 ....  
 ....  
 ....  
 ....  
 ....  
 ....

**Keywords**– Versió en anglès de les paraules clau. ....  
 ....



## 1 INTRODUCTION

**R**EINFORCEMENT 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 environment through 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. In the 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 achievement is AlphaGo, the model who beat the

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

European Go champion Fan Hui by 5 games to 0 [5].

Videogames provide an ideal environment 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 accomplished 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 my RL model from zero: Without using libraries.
4. Experiment with self-play training in a videogame: Using a ping pong environment, train the agents from both sides of the game.
5. Analyze and Compare every model: With empiric data.

## 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: Environment 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

- Phase 3: Deep Q-Network (DQN) applied to image-based game (04/05/25 - 25-05-25)
  - Task 9: Select environment and study the DQN algorithm
  - Task 10: Implement and train the agent
  - Task 11: Test the algorithm
  - Task 12: Document the implementations

- Phase 4: Self-play
  - Task 13: Study petting zoo library
  - Task 14: Implement and train two agents
  - Task 15: Test the agents implemented
  - Task 16: Train and test a different non-tabular algorithm in every agent
  - Task 17: Document the implementations

- Phase 5: Final inform and conclusions
  - Task 18: Document the 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. Environment — responds to the agent actions. It provides the states and the rewards, and changes based on the agent actions. States — Represent the actual environment situation by the perspective of the agent. Actions — The possible decisions 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  $\gamma$ , 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.

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

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

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 environment, would be the aliens, the visual scenery, 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 rewards: +1 for shooting an enemy, -1 for losing a life and 0 for not doing anything for example. The policy: 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 environment 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 environment 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 environment [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 environment. Learns based on experience.

The monte carlo algorithms works like this: The agent follows a policy [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 policy, 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 get a reward, the state value gets immediately updated

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.  $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma Q(s', a') - Q(s, a)]$ . Q-learning — is almost the same, but learns from the best action possible.  $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$

## 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. 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 THE GYMNASIUM LIBRARY

Gymnasium is a python library, forked from OpenAI Gym Library, that provides a standardized interface for creating environments 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 environments 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.

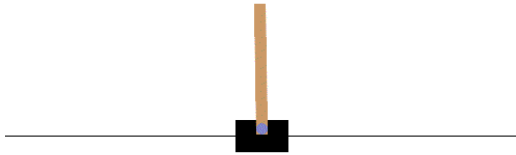


Fig. 2: CartPole-v1: A classic control problem with continuous states. The goal is to balance a pole on a moving cart.

## 5.1 Implementing tabular methods

### 5.1.1 Implementation of the algorithms

Three tabular methods were implemented and tested on the FrozenLake-v1 environment: Monte Carlo, SARSA and Q-Learning. FrozenLake has an important environment option: slippery. If slippery is true the game is not deterministic and the agent has a 30% chance to slide and choose a random action. All the same hyperparameters were used for every algorithm: 100000 episodes, 150 max steps of an episode, alpha ( $\alpha$ ) 0.1, gamma ( $\gamma$ ) 0.99, 1.0 initial epsilon, 0.9999 epsilon decay every episode, and the minimum epsilon 0.01. The environment was set to 8x8 grid with 0.8 probability of holes. The agent can choose from 4 discrete actions: left, right, up, down.

#### Monte Carlo

```
Initialize Q(s, a) arbitrarily
For each episode:
    Initialize empty list for (s, a, r)
    Initialize state s

    Repeat until episode ends:
        Choose action a using epsilon-greedy policy
        Take action a, observe reward r and next state s'
```

```
Append (s, a, r) to episode list
s = s'
```

```
For each (s, a) in the episode (first visit only):
```

```
    G = total discounted return after first occurrence
```

```
     $Q(s, a) = Q(s, a) + \alpha * [G - Q(s, a)]$ 
```

#### SARSA

```
Initialize Q(s, a) arbitrarily
```

```
For each episode:
```

```
    Initialize state s
```

```
    Choose action a using epsilon-greedy policy
```

```
    Repeat until episode ends:
```

```
        Take action a, observe reward r and next state s'
```

```
        Choose next action a' using epsilon-greedy policy
```

```
         $Q(s, a) = Q(s, a) + \alpha * [r + \gamma * Q(s', a') - Q(s, a)]$ 
```

```
        s = s'
```

```
        a = a'
```

#### Q-Learning

```
Initialize Q(s, a) arbitrarily
```

```
For each episode:
```

```
    Initialize state s
```

```
    Repeat until episode ends:
```

```
        Choose action a using epsilon-greedy policy
```

```
        Take action a, observe reward r and s'
```

```
         $Q(s, a) = Q(s, a) + \alpha * [r + \gamma * \max_{a'} Q(s', a') - Q(s, a)]$ 
```

```
        s = s'
```

Reward shaping was implemented because the original FrozenLake environment 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.

### 5.1.2 Results and Conclusions from Training and Evaluation

30000 episodes and 150 max steps were used for testing every algorithm. All the algorithms performed with good results in the deterministic environment achieving a 100% success on all the tests. Monte Carlo was a little bit slower in the training of deterministic environment and performed worse when using non-deterministic environment. Q-learning performed the best on non-deterministic achieving more than a 70% success. Monte Carlo performed poorly on the non-deterministic reaching just less than 35% success rate.

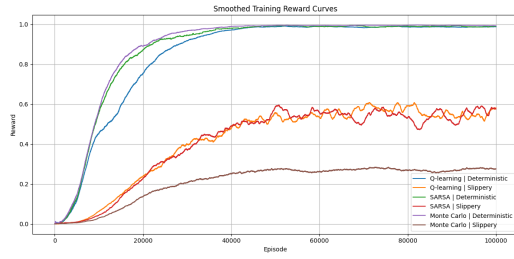


Fig. 3: Smoothed training reward curves

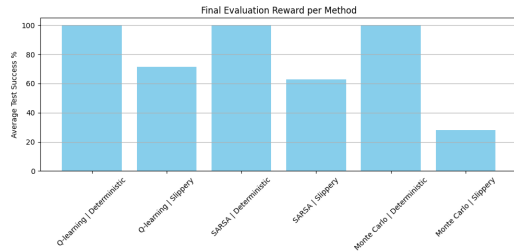


Fig. 4: Tested success (%)

## 5.2 Implementing non-tabular methods

### 5.2.1 Implementation of the algorithms

One non-tabular method was implemented on the CartPole-v1 environment: Deep Q-Network. The hyperparameters were: 500 episodes, 200 test episodes, 500 max steps of an episode in training and 1000 in testing, 64 batch size, gamma ( $\gamma$ ) 0.99, 1.0 initial epsilon, 0.995 epsilon decay every episode and minimum epsilon 0.01. 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 environment: 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.

#### DQN

```
Initialize Q-network with random weights
Initialize replay buffer B

For each episode:
    Initialize state s
    For each step of the episode:
        With probability epsilon:
            Choose random action a
        Else:
            Choose a = argmax_a Q(s, a)

        Execute action a, observe reward r
        and next state s'
        Store (s, a, r, s', done) in buffer B
        s = s'

    If buffer B has enough samples:
        Sample minibatch of transitions
        from B
        For each transition (s, a, r, s', done):
            If done:
                target = r
            Else:
```

```
                target = r + gamma *
                max_a' Q(s', a')

        Update Q-network to
        minimize:
            loss = (Q(s, a) -
            target)^2

        Decay epsilon: epsilon = max(
        min_epsilon, epsilon * decay_rate)
```

### 5.2.2 Results and Conclusions from Training and Evaluation

200 episodes and 1000 max steps were used for testing the algorithm. The DQN model performed good although it falls in training within the last episodes. 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. 5: Smoothed training reward curves

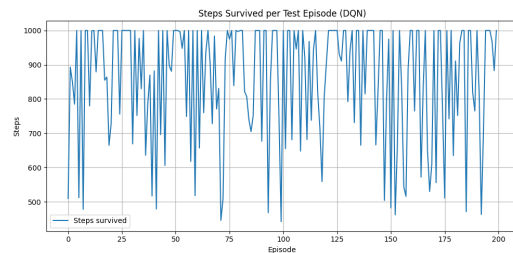


Fig. 6: Steps archived for every test episode

## 5.3 Deep Q-Network applied to image based game

### 5.3.1 Implementation of the algorithms

Pong from atari is a classic Atari 2600 game and it's the environment it will be used to train and evaluate a reinforcement learning dqn. The difference between this environment 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 84x84 in greyscale images and FrameStack of 4 is used (4 frames), to capture motion. Also FrameSkip of 4 is used (4 frames) to remove useless in-between frames. A deep neural network 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. It start with an epsilon value of 1.0 to 0.1.

### 5.3.2 Results and Conclusions from Training and Evaluation

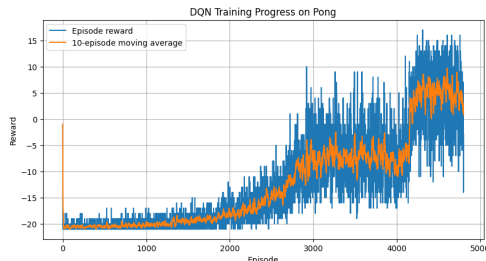


Fig. 7: Smoothed training progress

- [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.

## 6 APPENDIX

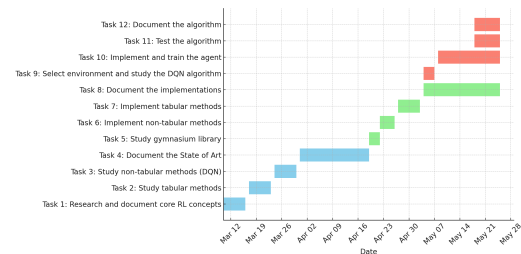


Fig. 8: Grantt diagram

## 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).