Analysis and Implementation of Tabular and Approximate Methods in Videogame Environments

Daniel García Castro

July 1, 2025

Resum– Aquest projecte consisteix en estudiar Reinforcement Learning (RL) mitjançant la implementació i anàlisi de diversos algorismes en entorns de videojocs. El projecte està dividit en dos blocs. El primer es sobre mètodes Tabulars, on s'implementen els algorismes Value-Iteration (Programació Dinàmica), First Visit Control (Monte Carlo) i Q-Learning (TD Learning). Aquests mètodes són implementats a l'entorn Frozen Lake i els seus resultats són analitzats i comparats. El segon bloc és sobre mètodes Aproximats, aquests són una combinació entre RL i Deep Learning. Els mètodes implementats són Deep Q-Learning i les seves variacions, els quals utilitzen Xarxes Neuronals Convolucionals per interpretar imatges. Aquests mètodes s'implementen al videojoc d'Atari Space Invaders. Els seus resultats s'analitzen i es comparen per determinar com afecta cada variació a l'entrenament i als resultats.

Paraules clau– Reinforcement Learning, Mètodes Tabulars, Mètodes Aproximats, Programació Dinàmica, Value-Iteration, Monte Carlo, Temporal Difference Learning, Q-Learning, Xarxa Neuronal, Deep Q-Network

Abstract– This project consists in studying Reinforcement Learning (RL) through the implementation and analysis of different kinds of algorithms in videogame environments. The project is divided in two blocks. The first is about Tabular methods, where there are implemented the methods Value-Iteration (Dynamic Programming), First Visit Control (Monte Carlo) and Q-Learning (TD Learning). These three are implemented in the Frozen Lake environment and their results are analyzed and compared. The second block is about Approximate methods, which are a combination of RL and Deep Learning. The methods implemented are Deep Q-Learning and its variations, which utilize Convolutional Neural Networks to be able to interpret images. These are implemented in the Atari game Space Invaders. Their results are analyzed and compared to see how each variation affects the training process and the results obtained.

Keywords– Reinforcement Learning, Tabular Methods, Approximate Methods, Dynamic Programming, Value-Iteration, Monte Carlo, Temporal Difference Learning, Q-Learning, Neural Network, Deep Q-Network

1 Introduction

Reinforcement learning (RL) is a field of machine learning where an agent learns to make decisions in an environment to maximize reward. Supervised learning relies on labeled examples to guide the training process, while unsupervised learning seeks to uncover hidden structures or patterns within the data without any labels. In contrast, RL takes a different approach, an agent interacts with its environment and learns by trial and error. Each time the agent takes an action, it receives a reward signal, and over time it discovers which actions yield the highest cumulative reward and prioritizes them.

In the case of videogames, the environment is the game screen, and the agent is the playable character. The agent chooses actions, and depending on what these cause, the environment provides a reward.

Positive rewards can be obtained by beating a level or

• Contact E-mail: daniel.garciacastr@autonoma.cat
• Specialization: Computing
• Work tutored by: Jordi Casas Roma (Computing)
• Course: 2024/25

obtaining points in the game, but in some cases the environment may also penalize the agent with negative rewards if it looses the level or if gets hit by an enemy for example. This way through trial and error the agent will learn which actions it should prioritize in order to beat the game or score the most points, depending on what gives the highest reward.

2 OBJECTIVES

The goal of this project is to implement various RL Algorithms on videogame environments and compare the obtained results to determine their strength and flaws in practical cases.

Starting with tabular methods to solve simple game environments and ending in Deep RL Methods to solve more complex Atari games. The individual objectives are the following:

1. Gain an understanding of RL and its algorithms as well as familiarize myself with the Gymnasium library [1] in Python
2. Implement three different tabular methods solving a simple environment, visualize the results obtained and compare them. These methods are Dynamic Programming, Monte Carlo and Q-learning
3. Implement the method Deep Q-Learning [2] and its variants in an Atari game using the Python library ALE [3], visualize and compare their results.
4. Implement the method REINFORCE [4] in an Atari game and compare its results against Deep Q-Learning.

3 METHODOLOGIES

I have decided to use the Agile methodology. This methodology will let me divide the project into multiple sprints, usually 2 weeks long each. During these sprints I will completely focus on the implementation of a concrete algorithm. After each sprint I will evaluate it to determine if it has gone as planned and if there should be any changes to future sprints. To keep track of the sprints and to be able to modify them I am using Microsoft Project. This tool allows me to visualize a Gantt diagram, shown in Fig. 8 in Appendix A.1, with the sprints and its dates, and lets me modify them or add subtasks during the development of the project if needed.

4 PLANNING

This project will be divided into two main blocks:

Tabular Methods:

1. Dynamic Programming (2 Weeks)
2. Monte Carlo (2 Weeks)
3. Temporal Difference Learning (2 Weeks)
4. Model Comparison (1 weeks)

Approximate Methods:

1. Deep Q-Learning (5 weeks)
2. Model Comparison (2 weeks)

There will also be another two weeks after the Approximate Methods block to prepare the final report and the presentation.

5 INTRODUCTION TO RL

As explained in the introduction, RL does not learn through labeled data, but through trial-and-error interacting with the environment as shown in Fig. 1.

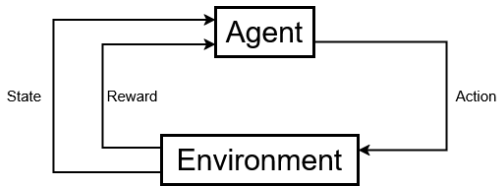


Fig. 1: RL Diagram

We have the following components:

1. Agent: The agent interacts with the environment doing specific actions and learns receiving rewards from it.
2. Environment: The environment represents what the agent can interact with, depending on those interactions the environment will change its state and send it to the agent alongside a reward.

And the following signals:

1. Action: The decision taken by the agent, for example an action can be moving in a specific direction.
2. Reward: Depending on the action chosen the environment will grant a reward, for example if the action makes the agent reach the goal it will be granted a positive reward.
3. State: It represents the current situation of the environment. It can change for example if the agent moves, changing its position and creating a new situation.

5.1 Markov Decision Process

RL problems are often modeled as a Markov Decision Process (MDP), which provides a mathematical framework for decision-making in environments where outcomes are partly random and partly under the control of an agent. An MDP is defined by a set of all possibles states and actions as well as:

1. Transition probability: the probability $P(s' \mid s, a)$ of transitioning to state s' after taking action a in state s.
2. Reward function: a function $R(s, a, s')$ that gives the reward received when moving from state s to state s' via action a.

3. Discount factor (γ): a value between 0 and 1 that determines the importance of future rewards. A lower value makes the agent short-sighted, while a value close to 1 encourages long-term planning.

In an MDP, the outcome of an action depends only on the current state and not on the sequence of states that preceded it. The goal in an MDP is to obtain a policy $\pi(s)$ that maps each state to an action maximizing the cumulative reward (return) Eq. 1 which is the sum of the rewards obtained by all future actions.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^T R_{T+t+1} \quad (1)$$

5.2 Bellman Equation

The Bellman Equation is a key concept in RL used to calculate the value of a state or a state-action pair based on expected future rewards.

$$V(s) = \sum_{s',r} p(s', r \mid s, \pi(s)) \left[r + \gamma V(s')\right] \quad (2)$$

$$Q(s,a) = \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma \max_{a'} Q(s', a')\right] \quad (3)$$

In Eq. 2, the value of a state s is computed as the expected reward r received by following the policy $\pi(s)$, plus the discounted value of the next state s'. The value obtained means how good is it to be in state s when following the policy.

Eq. 3 estimates the value of taking action a in state s by looking at the expected reward and the best possible future return from the next state. The result obtained means how good is it to take action a while being on state s.

5.3 Types of RL Methods

There are two main types of RL methods, value-based and policy-based.

Value-based methods focus on estimating value functions, such as $V(s)$ or $Q(s,a)$, to determine how good it is to be in a state or to take an action. The policy is then derived indirectly by selecting the action with the highest estimated value.

Policy-based methods, on the other hand, try to learn the policy directly without relying on a value function. These methods optimize the policy by increasing the likelihood of actions that lead to higher rewards.

6 State of the Art

This section provides an overview of the main categories of RL methods, with a focus on those that are relevant to or implemented in this project.

6.1 Tabular Methods

In RL, tabular methods refer to techniques that represent and update the value of states or state-action pairs in a table format.

These methods use a table that stores the expected rewards for each state or action in the environment. As the agent interacts with the environment, the values in the table are updated based on the rewards obtained, the goal is to learn the optimal policy by refining the values in the table.

6.1.1 Dynamic Programming

Dynamic Programming (DP) [5] methods require a complete model of the environment, including the transition probabilities and rewards. These methods solve reinforcement learning problems by using Bellman equations to compute value functions.

DP algorithms iteratively update value estimates across all states, assuming full knowledge of how actions lead to next states and rewards. This makes them suitable for planning rather than learning from direct experience.

There are two main methods in DP, the first is Value-Iteration, which is used to compute the optimal value function for each state in an MDP, the Bellman Equation Eq. 2 is used to update the value of each state until it converges to the optimal value-function, after that we can extract the optimal policy by selecting the action that maximizes the value function. A pseudo code for Value-Iteration can be seen at Algorithm 1.

The second is Policy-Iteration, which is used to obtain the optimal policy, it is done by alternating two steps, Policy Evaluation to obtain the value function for a given policy, and then Policy Improvement Eq. 4 to update the policy to an improved one by selecting the action that maximizes the return.

$$\pi(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma V(s')\right] \quad (4)$$

6.1.2 Monte Carlo

Unlike DP, knowledge of the transition probabilities and rewards isn't required for Monte Carlo [5] methods, because they learn from experience through trial-and-error.

Monte Carlo methods estimate value functions and improve policies by sampling complete episodes, after each episode, the agent updates its estimates based on the return shown in Eq. 1. A pseudo code for a Monte Carlo implementation can be seen in Algorithm 2.

6.1.3 Temporal Difference Learning

Temporal Difference (TD) [5] Learning methods learn by trial-and-error like Monte Carlo, but the main difference is when the value estimates are updated and how they are calculated. Instead of updating the value estimates at the end of an episode, TD methods update them at every step, using the reward received and the estimated value of the next state with the Eq. 5.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big] \quad (5)$$

TD learning methods make use of a Q-Table, this table contains a value for each state-action combination in the environment, the highest value in state represents the best action to take when the agent is there. There are two main

methods, Q-Learning and SARSA. Q-Learning is an off-policy method, meaning it updates the Q-Table using the maximum possible reward of the next state, regardless of the action actually taken. In contrast, SARSA is an on-policy method, it updates the Q-Table based on the action the agent actually chooses in the next state, following its current policy. A pseudo code for Q-Learning can be seen in Algorithm 3.

Algorithm 1 Value Iteration

Initialize $V(s)$ arbitrarily for all states s
repeat
  $\Delta \leftarrow 0$
  for all state s **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s'} P(s' \mid s, a)[R(s, a, s') + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  end for
until $\Delta < \theta$

Algorithm 2 Monte Carlo Control First Visit

Initialize:
  $Q(s, a) \leftarrow 0$ for all states and actions
  $N(s, a) \leftarrow 0$ for all states and actions
  $\pi(s) \leftarrow 0$ for all states
  $\epsilon \leftarrow 1$
for each episode **do**
  Generate episode:
    With probability ϵ: choose random action
    Else: follow policy π
  Store (s, a, r) in `episode_log`
  Decrease ϵ linearly by the number of episodes
  $G \leftarrow 0$
  for t in `reversed(range(episode_log))` **do**
    $(s, a, r) \leftarrow$ `episode_log(t)`
    $G \leftarrow r + \gamma G$
    if (s, a) is the first visit in the episode **then**
      $N(s, a) \leftarrow N(s, a) + 1$
      $Q(s, a) \leftarrow Q(s, a) + \frac{1}{N(s,a)}(G - Q(s, a))$
      $\pi(s) \leftarrow \max(Q(s))$
    end if
  end for
end for

Algorithm 3 Q-Learning

Initialize:
  $Q(s, a) \leftarrow 0$ for all states s and actions a
for each episode **do**
  Restart environment and get initial state s
  while not terminated and not truncated **do**
    Choose action a:
      With probability ϵ: $a \leftarrow$ random action
      Else: $a \leftarrow \arg\max Q(s, a)$
    Take action $a \rightarrow$ observe reward r and next state s'
    Update $Q(s, a)$:
      $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
    Set $s \leftarrow s'$
  end while
  Decrease ϵ linearly by the number of episodes
  Decrease α linearly by the number of episodes
end for

6.2 Approximate Methods

Approximate methods instead of using tables they replace them with parameterized functions that estimate value functions or policies, for example, in Q-learning a Q-table is used to store the state action pairs, however in deep Q-Learning Neural Networks (NN) are used instead.

This change lets us take on more complex environments like Atari games, Tabular Methods are not useful in these environments because of the size of the observation space. Since the Atari games states are determined by images it would require an enormous amount of memory and time to fill a table containing every possibility, that's why instead of tables non-linear functions are used to determine the best action.

There are a lot of different approximate methods, they can be divided in Value-Based methods Policy-Based Methods and Actor-Critic methods. They are divided in these subgroups depending on if they approximate value functions, policies or do a mix or both. Although there are a lot of different methods, the explanation will be focused on the main Deep RL methods since those are the ones relevant to the ones implemented in this project, these being Deep Q-Learning, Policy Gradient methods and Actor-Critic methods.

6.2.1 Deep Q-Learning

Deep Q-Learning (DQL) [2] is one of the most powerful Value-Based method and it is the result of combining Deep Learning and Q-Learning. When there is a large number of states and actions it becomes impossible to store them in a table so in order to map them, instead we use non-linear functions, taking advantage of Deep Learning techniques like NNs to represent the Q-table. A pseudo code of the DQN algorithm can be seen in Algorithm 4.

Even though this might sound simple, there are some important complications when applying this method, mainly due to how data is collected and used during training. Below are two problems and how to address them:

1) Sequential Data:

One of the main issues comes from the fact that the data the NN receives is sequential. For example, in a videogame environment, the agent first receives a state where the character is in a certain position, then chooses an action based on that. The next state it receives is the direct result of that action. This means that consecutive states are very similar to each other. If we were to train the network using these states in the same order they're generated, we'd be feeding it highly correlated inputs step after step.

After each interaction with the environment, we update the Q-values. But because the states are so highly correlated, each new update overrides the last one. This constant overwriting makes it hard for the agent to retain what it learned from earlier experiences, making it forget actions it previously took.

Solution:

Experience Replay is the technique used to solve this. Instead of training immediately after each step, the full expe-

rience (state, action taken, reward received, next state, and whether it's terminal) is saved into a buffer. Later, a random batch of experiences is sampled from this buffer to train the network. This breaks the sequence and introduces more variety in the training data, which helps the model generalize better avoiding training on sequential data.

2) High Correlation Between States and Targets:

The other problem is how target values are calculated during training. When updating the Q-values, the same NN is used to estimate both the current value and the target value. Since both come from the same network and it's constantly changing, the target itself shifts every time we update, which makes learning unstable.

Solution:

To avoid this problem a second NN is implemented. One of the two networks will be used to calculate the target value (Target Network) and the other to calculate the predicted value (Policy Network). Both NN will be initialized with the same values, however the Policy Network will be updated after each step while the Target Network remains the same, after N steps the Target Network will copy the values in the Policy Network. Thanks to this difference in updates in the NN the correlation between states breaks.

6.2.2 Policy Gradients

Policy Gradients [5] similarly to Deep Q-Learning are a combination of Deep Learning and Reinforcement Learning, however here instead of learning a value function and deriving the policy form it here the policy is learned directly optimizing it through gradient ascent while interacting with the environment. One of the most important Policy Gradient methods is REINFORCE [4], this method is a Monte Carlo method that interacts with the environment and after the episode ends it computes the return, with that return it performs a policy gradient Eq. 6 to increase the probability of the actions that led to high return.

$$\nabla J(\theta) = E_\pi \left[G_t \nabla_\theta \log \pi(A_t \mid S_t; \theta) \right] \tag{6}$$

Then it updates the policy parameters which can be represented as weights in a NN by adding the policy gradient with the learning rate to it. Then it repeats the process to the next episode, by doing this it optimizes the policy in order to maximize the return. These methods can be really powerful however just like DQN there are also problems that affect them and need to be solved by adding modifications, these methods can suffer from the same correlation problems that DQN faced as well as a lot of variability in its results depending on the rewards the agent obtains. To try to solve these problems the Actor-Critic methods appeared.

6.2.3 Actor-Critic

The purpose of Actor-Critic Methods is to combine the best of policy-based and value-based methods. The idea is to divide the model into an Actor and a Critic, this is done with two NNs, one to represent each. The actor decides what action to take, so its approximation function is based on the policy, and the Critic evaluates how good the action taken by the Actor was, so its function approximates the Q-value for the action. The evaluation of the Critic is used to calculate the advantage that the action taken by the Actor had compared to the average and then update the policy parameters based on that. During the training loop we alternate on updating the Critic to evaluate better the performance of the Actor and updating the Actor to upgrade its policy. The main methods are A2C (Advatage Actor-Critic) [6] which divides the Advantage obtained in two layers of the NN between the action value and the state value, by doing that it reduces the variance in the training. The other is A3C (asynchronous Advantage Actor-Critic) [6] which on top of what A2C did it makes use of multiple asynchronous agents that individually give information to the Critic and periodically update its weights based on the Critic.

Algorithm 4 DQN

Initialize:
  Policy Network Q with random weights θ
  Target Network Q' with weights $\theta' \leftarrow \theta$
  Replay Buffer D empty
for each episode **do**
  for each step **do**
    In state s choose action a:
      With probability ϵ: $a \leftarrow$ random action
      Else: $a \leftarrow \arg\max Q(s)$
    Store (s, a, r, s') in D
    Get mini batch of experiences from D
    for each (s, a, r, s') in mini_batch **do**
      if episode ended **then**
        $y_j \leftarrow r_j$
      else
        $y_j \leftarrow r_j + \gamma \max_{a' \in \mathcal{A}} \hat{Q}(s', a'; \theta^-)$
      end if
    end for
    Calculate loss
    Update Q with SGD
    Sync Networks every N steps ($\theta' = \theta$)
  end for
end for

7 Frozen Lake

All the code implemented in this project can be found in this GitHub repository: `https://github.com/DaniGarcia02/RL_In_Videogames`.

To evaluate the different tabular methods the environment used has been Frozen Lake, it can be seen in Fig. 2, the objective is to reach the goal without falling in the holes.

Fig. 2: Frozen Lake

These are the actions the agent can make:

1. Move Up
2. Move Left
3. Move Right
4. Move Down

These are the rewards the environment gives:

- Reach Goal: +1
- Any other square: 0

I have trained this algorithm with the 'slippery flag' of gymnasium, what this does is make that every time the agent moves there's a 2/3 chance of it slipping sideways to a square it didn't intend to go. For example if it moves down there is a 1/3 chance of moving right and a 1/3 chance of moving left instead of down. This makes the environment a lot more complex and lets us see the robustness of the algorithms against situations where the agent can't always go where it wants.

I have tested every method in two different Frozen Lake maps, one is the 4x4 shown in Fig.2, and the other is an 8x8 map which will make learning harder since the paths to the goal will be longer and the agent won't move in the direction it wants more times than in the smaller map.

7.1 Value-Iteration Results

The Parameters used in Value-Iteration are:

- Discount Factor (γ): 1
- Stopping Threshold (θ): 1e-10

←	↑	↑	↑
←	None	←, →	None
↑	↓	←	None
None	→	↓	G

Fig. 3: Optimal Policy in 4x4 map

The policy shown in Fig. 3 is the one obtained in the 4x4 map with Value-Iteration, this is the optimal policy in this environment, it takes 0.04 seconds to train. Testing it in 1000 episodes it reaches around 85% accuracy. It is impossible to reach 100% accuracy in this map due to the agent slipping towards the square that's between two holes, there's no action that guarantees safety 100% of the times there so sometimes it won't reach the goal.

In the 8x8 map the policy obtained is shown in Fig. 9 in Appendix section A.2. Value-Iteration also obtains the optimal policy for the map, it is able to reach the goal 100% of the time since there is a path that always avoids falling and still reaches the goal. The time it takes to train is 0.3 seconds.

7.2 Monte Carlo Results

The Monte Carlo algorithm uses ϵ-greedy, which means that with probability ϵ the agent will choose a random action instead of following the policy, this value will gradually decrease during training. The objective of this is to make the agent explore by choosing a lot of random actions at the start and gradually making it exploit more to optimize the best path to end.

The Parameters used in Monte Carlo are:

- Discount Factor (γ): 1
- Epsilon (ϵ): Starts at 1 and linearly decreases until it reaches 0.05 by the last episode
- Number of episodes: 350,000 in the 4x4 map and 2,000,000 in the 8x8 map

←	↑	←	↑
←	None	←	None
↑	↓	←	None
None	→	↓	G

Fig. 4: Policy obtained by Monte Carlo in the 4x4 map

This is the policy obtained in the 4x4 map, its slightly different that the one obtained in value-iteration, it is less optimal because the value in the first row and the third column can lead to the square between two holes, and as mentioned before there is no action that always avoids falling there. The accuracy is around 75% and it takes around 75 seconds to train.

The policy obtained in the 8x8 map is shown in Fig. 10 in Appendix section A.2. In this map the accuracy obtained is around 90% and it takes around 1,150 seconds to train.

7.3 Q-Learning Results

Q-Learning also uses ϵ-greedy, but it also decreases the learning rate (α) during training, the objective of this is to take bigger leaps during early training and as the agent starts exploiting more it reduces to take smaller leaps, that way making smaller adjustments to the policy that follows the best path found.

The Hyperparameters used in Q-Learning are:

- Discount Factor (γ): 0.99
- Epsilon (ϵ): Starts at 1 and linearly decreases until it reaches 0.05 by the last episode
- Learning Rate (α): Starts at 0.7 and linearly decreases until it reaches 0.01 by the last episode
- Number of episodes: 10,000 in the 4x4 map and 200,000 in the 8x8 map

The policy obtained in the 4x4 map is the same as in Value-Iteration, so the optimal policy. It has around 85% accuracy and it takes around 2s to train. The policy obtained in the 8x8 map is shown in Fig. 11 in the Appendix in section A.2. The Accuracy obtained is around 90% and it takes around 110s to train.

7.4 Comparing the models

Comparing the results of the models it's clear that the best algorithm for Frozen Lake is Value-Iteration, it is by far the fastest and it reaches the optimal policy in both maps. The inconvenient of this method is that it requires complete knowledge of the environment so it's less versatile as the other two methods, it works in less situations.

Comparing now Monte Carlo and Q-Learning the best method is Q-Learning. It reaches similar results than Monte Carlo but it's way faster, this is likely a result of how the rewards are distributed in Frozen Lake, since the only reward is at the goal and Monte Carlo only learns by the return, until an episode reaches the end it does not learn. While Q-Learning updates its policy after each step making this reward scarcity less of a problem and learning much faster.

Method	4x4 Map		8x8 Map	
	Acc.	Time	Acc.	Time
VI	85%	0.04s	100%	0.4s
MC	75%	73s	90%	1150s
QL	85%	2s	90%	110s

TABLE 1: RESULTS OF THE TABULAR METHODS

8 SPACE INVADERS

To test out the approximate methods I will use the Space Invaders Atari game, the objective in Space Invaders is to shoot all aliens before they get to the bottom of the screen while avoiding being shot.

This environment, shown in Fig. 5, has a much bigger observation space than Frozen lake, this is because while Frozen Lake had a small number of squares that determined the state, here the states are determined by images that are 210 x 160 x 3 pixels, due to this size increase in the environment Tabular methods are no longer an option because it would require an enormous amount of memory and time to fill a table determining the best possible action for every state.

This environment has a value called 'repeat action probability', what it does is that with a specified probability the action the agent takes will be the same as the one chosen in the last step. This makes the agent not always act in the way it intends, this makes the environment harder but also lets us see the robustness of the policy. The value this flag has had in this implementation is 0.25, which is the default in Space Inaders v5 of ALE [3].

Fig. 5: Atari Space Invaders

These are the actions the agent can do:

1. No Operation
2. Move Left
3. Move Right
4. Fire
5. Move Left and Fire
6. Move Right and Fire

These are the rewards that can be obtained in the environment, they are obtained when successfully shooting an alien.

Rewards (From bottom to top):

- Aliens in Row 1: +5
- Aliens in Row 2: +10
- Aliens in Row 3: +15
- Aliens in Row 4: +20
- Aliens in Row 5: +25
- Aliens in Row 6: +30
- Command Ship: +200

8.1 Deep Q-Learning

In this section there's the implementation of a DQN with Convolutional Neural Networks (CNN) [7] and the results obtained. The implementation is made using the PyTorch library [8] in Python and varies a bit from what was explained in the state of the art. There are many variations of DQN that can improve its results, here's the explanation of the ones implemented in this project.

8.1.1 Neural Network Structure:

Before getting to the variations of DQN here's the explanation of the network structure used in this project, the NN consists of three Convolutional layers and two fully connected layers. The objective of the Convolutional layers is to extract features from the frames in space invaders, each layer connects to the next and after the last of the three its output is flattened and passed to the fully connected layers. Their objective is to interpret the features extracted by the previous layers and map them into actions in the environment. A diagram of the DQN structure is shown in Fig. 6

The first layer receives the image directly, in this case its 4 frames that have been stacked, and resized to be 84 x 84 pixels, this way the agent would have more information like the moving shots and aliens, but also, I have reduced its size to make the execution faster. The images are in grayscale to reduce furthermore the input size since having 3 channels for color would triplicate the size of the input. This image goes through the layers of the NN and the output of the last layer is the action the agent should take.

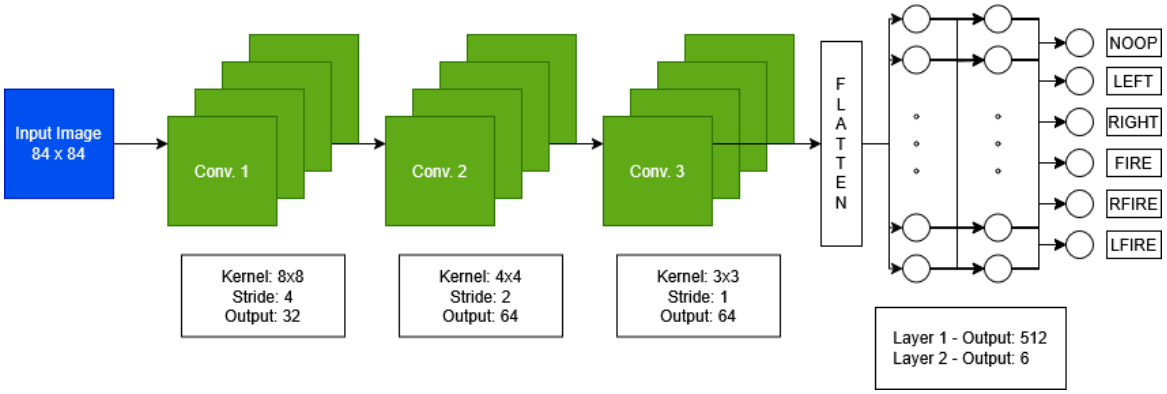


Fig. 6: Diagram of the DQN structure

8.1.2 Double DQN:

The first variation is Double DQN [9], the change is in the way the target value is calculated.

$$\text{DQN:} \quad y = r + \gamma \max_a \hat{Q}(s', a^-; \theta^-)$$
$$\text{DDQN:} \quad y = r + \gamma \hat{Q}\big(s', \arg\max_a Q(s', a; \theta); \theta^-\big)$$

In the original DQN the target value is obtained by the target network $\hat{Q}$, however we have no guarantee that the action chosen is the one that gives the best value in the policy network Q, this causes the DQN to overestimate the Q-values.

The new formula helps by letting the policy network Q choose the action and then the target network $\hat{Q}$ uses that action to get the target value. This equation is the Bellman Equation for the target value but with the addition of θ and θ^- which represent the NN weights of the policy and target networks respectively.

8.1.3 Dueling DQN:

Another variant implemented is a Dueling DQN [10]. This variant consists in changing the structure of the NN. In the original DQN a single layer is used to get the Q-value, the change that makes the Dueling DQN is that instead of directly giving the Q-value it divides the layer that gives Q-values in two, one for getting the value of being in the current state and one for getting the value of choosing an action and changing state, then this two are combined to obtain the Q-value. This allows the network to learn which states are relevant and which are not, avoiding useless changes that could impact negatively to the learned policy.

8.1.4 Prioritized Experience Replay (PER):

PER [11] is another variant of DQN, the change it makes is that now the experiences inside the buffer are 'ranked', we will add to the buffer the importance of the experience using this formula:

$$P_t = |\delta_t| + \varepsilon \tag{7}$$

This is the TD error with a constant ϵ that ensures all experiences have a chance of being selected, there is also a value α that determines the probability of the best rewards being chosen. However, implementing just this will create a bias in the experiences chosen, to avoid this we also have to introduce importance sampling to adjust the weights of the most viewed experience. Every experience has a weight determined by the following formula:

$$w_i = \left(\frac{1}{N\, P(i)}\right)^{\beta} \tag{8}$$

Here N represents the size of the buffer, P(i) the probability distribution and β is a hyperparameter that works in a similar way to ϵ in ϵ-greedy, it determines how much the weights affect the training and it starts at a small value and is increased towards 1 as the training progresses, like ϵ in ϵ-greedy but the other way around.

8.1.5 N-Step:

The objective of N-step [5] is to reduce the learning time by updating the values of the NN with the values of N steps. The Bellman equation would look like Eq. 9 if N had a value of 2.

$$Q(S_t, A_t) = R_t + \gamma R_{t+1} + \gamma^2 \max_{a'} Q(S_{t+2}, A_{t+2}) \tag{9}$$

Looking back at the equation shown in Eq. 3 the difference is that it computes 2 steps at once instead of one. If N were to have value 3 it would do 3 steps at once and so on. By increasing the number of steps computed every update the updates become more meaningful. However Eq. 9 is assuming that the action chosen is optimal, but that won't always be the case, so increasing the value of N also makes it more likely to update NN with suboptimal values leading to undesirable results. In the end N just becomes a new hyperparameter to tune which can lead to a better performance for the DQN.

8.1.6 Noisy Network:

Another variation implemented in this project is the Noisy Network [12], this method replaces ϵ-greedy. Noisy Networks introduce learnable noise directly into the weights of the linear layers of the NN. Instead of selecting actions randomly, the randomness is embedded in the network itself. During training, the weights are perturbed by noise, allowing the agent to explore different actions naturally as part of the forward pass. The new noisy layer weights can be represented like in Eq. 10.

$$w = \mu + \sigma \otimes \varepsilon \tag{10}$$

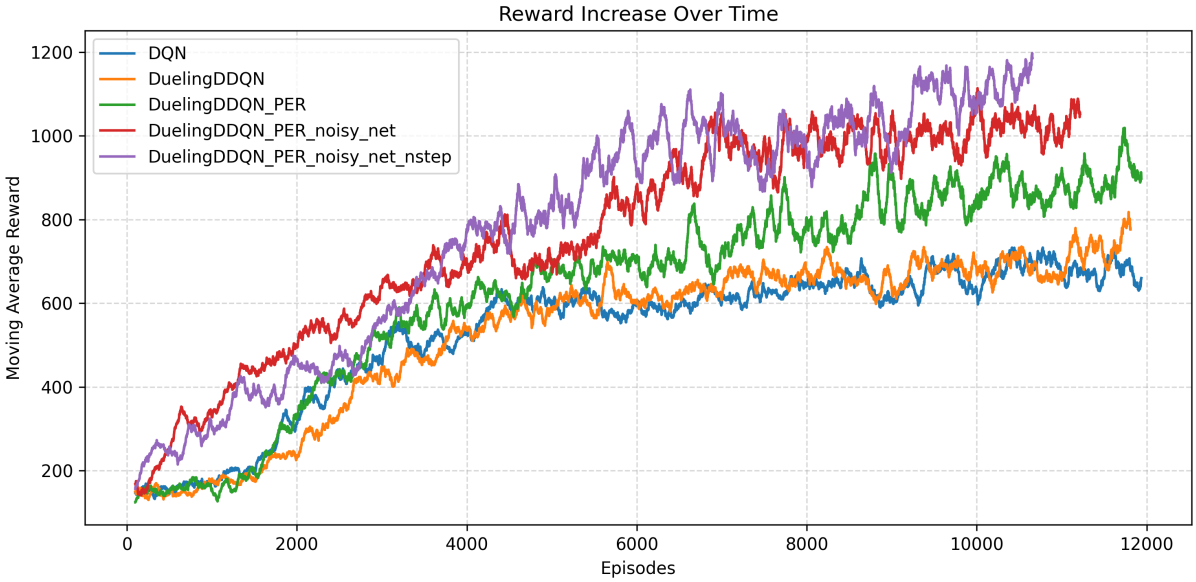


Fig. 7: Comparison of rewards in DQN variants

Here, μ and σ are learnable parameters, and ϵ is random noise sampled from a factorized Gaussian distribution. This approach allows the network to learn how much noise should be injected into each weight during training, controlling how much the agent explores in each state.

Unlike ϵ-greedy, this technique adapts the amount of exploration automatically during training and removes the need for manually scheduling epsilon decay and making exploration more state-dependent.

8.1.7 Results:

General DQN Hyperparameters:

- Total Number of Steps: 12,000,000
- Replay Memory Size: 400,000
- Learning Rate: 2.5×10^{-4}
- Mini Batch Size: 256
- Network Sync Rate: 1,000
- Discount Factor: 0.99

Hyperparameters without Noisy Network:

- Epsilon (ϵ): Starts at 1 and exponentially decreases until it reaches 0.01 by the last step

Hyperparameters for PER:

- Alpha (α): 0.6
- Beta (β): Starts at 0.4 and linearly increases to 1 during the first 1,000,000 steps

Hyperparameters for N-Step:

- Number of steps (N): 3

Hyperparameters for Noisy Network:

- Std initializer (σ): 0.5

Fig. 7 presents the average reward in the original DQN and all implemented variants. As it can be seen, DQN and DuelingDDQN are almost identical, this shows that either these variations don't really have much of an effect in Space Invaders or that there would need to be more episodes for the difference to be more noticeable.

Where the results start to be noticeably better is with the addition on PER, these results make clear that prioritizing the experiences chosen in training does have a great impact and that modifying the NN structure is not the only way to obtain better results.

This first three methods were using ϵ-greedy, and it can be seen in the graph that the reward increase roughly translates to the ϵ-greedy graph shown in Fig. 12 in Appendix section A.3. Because when epsilon is still high they grow consistently and when it starts getting closer to 0.01 the rewards start increasing really slowly almost stagnating. However this is only really true for the non-PER methods, when PER is added even though it still uses ϵ-greedy the rewards still increase at a considerable rate. These methods especially the non-PER ones would benefit from a longer training time since they are bound by epsilon.

The implementations with Noisy Network have ended with the best results, this shows how letting the networks adapt how much exploration the states need is more efficient that just starting with high exploration and reducing it. As it can be seen in Fig. 7 the methods with Noisy layers start learning faster since they don't only explore during the initial steps, also they don't stagnate as heavily as the other implementations since they can keep exploring if needed.

The best implementation over all is the one that combined every variation explained in this paper, even though the addition N-Step did not make a difference as big as Noisy and PER it still has let the network arrive at a policy that reaches 1,200 points as an average score. This implementation almost doubles the points of the simple DQN implemented which got a little over 600 points.

9 Conclusions

During this project I have been able to implement and test a lot of RL algorithms. From the simplest Tabular methods interacting in basic environments to more complex Deep RL methods that make use of Neural Networks and interact with complex environments like Atari games.

In regards of the Tabular method part of the project I have accomplished my objectives of implementing and testing three different kinds of methods and compared them, I was able to obtain results I'm proud of.

In the Deep RL part I was not able to accomplish everything I wanted, my initial idea was to implement and test the three different kinds of methods I described in the State of the Art but I underestimated how complex and time consuming this methods are. In the end I focused on DQN because there were a lot of modifications that could be made to improve the base algorithm that I ended up wanting to implement and compare those modifications and I didn't have time to focus on the other two methods.

In the end Reinforcement Learning is a really interesting and complex field that through this project I've been able dig into and see its potential and its applications, it also has helped me gain experience that will be useful not only for RL but also for other Artificial Intelligence fields.

10 Future Work

As for ways to improve this project in the future, it would be interesting to implement a Policy Gradient method like REINFORCE and an Actor-Critic method like A3C and then compare how they fare against the DQN implemented in this project.

It also would be interesting to test these methods in other environments outside Space Invaders to see how the different DQN variations affect other game environments different than Space Invaders.

Acknowledgements

I want to thank my tutor Jordi Casas Roma for introducing me to this field of Artificial Intelligence. It's a really interesting field that isn't touched on in the Computer Engineering degree and if it wasn't for this project I don't know if I would have ever done a project in it. I also want to thank my family for supporting me since its thanks to them that I'm able to study something I like.

References

[1] Gymnasium Documentation. [Online]. Available: https://gymnasium.farama.org/ (Accessed: June 26, 2025)

[2] Mnih, V., Kavukcuoglu, K., Silver, D., et al. (2013). Playing Atari with Deep Reinforcement Learning. arXiv. https://doi.org/10.48550/arXiv.1312.5602

[3] Arcade Learning Environment (ALE) Documentation. [Online]. Available: https://ale.farama.org/ (Accessed: June 26, 2025)

[4] Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine Learning, 8, 229–256. https://doi.org/10.1007/BF00992696

[5] Sutton, R. S., Barto, A. G. (2020). Reinforcement Learning: An Introduction (2nd ed.). [Online]. Available: http://incompleteideas.net/book/RLbook2020.pdf (Accessed: June 26, 2025)

[6] Mnih, V., Badia, A. P., Mirza, M., et al. (2016). Asynchronous Methods for Deep Reinforcement Learning. arXiv. https://doi.org/10.48550/arXiv.1602.01783

[7] Strobelt, H., Zablotskaia, P., Xu, M. (2020). CNN Explainer: Learning Convolutional Neural Networks with Interactive Visualization. [Online]. Available: https://poloclub.github.io/cnn-explainer/ (Accessed: June 26, 2025)

[8] PyTorch Documentation. [Online]. Available: https://pytorch.org/docs/stable/index.html (Accessed: June 26, 2025)

[9] Hasselt, H. V., Guez, A., Silver, D. (2016). Deep Reinforcement Learning with Double Q-learning. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16). arXiv. https://doi.org/10.48550/arXiv.1509.06461

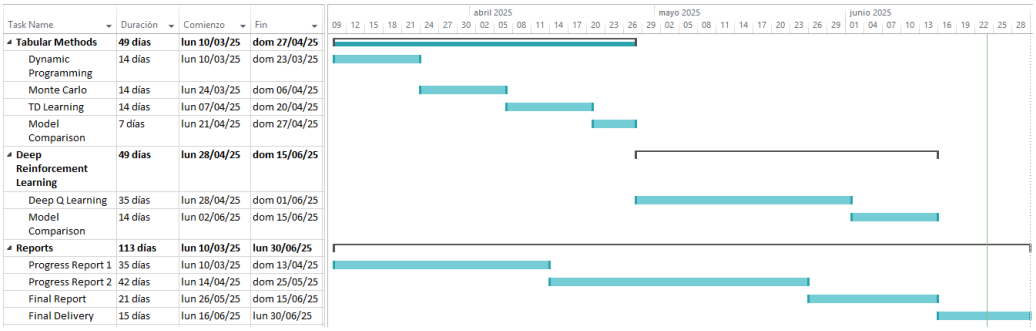
[10] Wang, Z., Schaul, T., Hessel, M., et al. (2016). Dueling Network Architectures for Deep Reinforcement Learning. In Proceedings of the 33rd International Conference on Machine Learning (ICML 2016). arXiv. https://doi.org/10.48550/arXiv.1511.06581

[11] Schaul, T., Quan, J., Antonoglou, I., Silver, D. (2015). Prioritized Experience Replay. arXiv. https://doi.org/10.48550/arXiv.1511.05952

[12] Fortunato, M., Azar, M. G., Piot, B., et al. (2018). Noisy Networks for Exploration. In Proceedings of the International Conference on Learning Representations (ICLR 2018). arXiv. https://doi.org/10.48550/arXiv.1706.10295

A Appendix

A.1 Methodology



Task Name	Duración	Comienzo	Fin
Tabular Methods	**49 días**	**lun 10/03/25**	**dom 27/04/25**
Dynamic Programming	14 días	lun 10/03/25	dom 23/03/25
Monte Carlo	14 días	lun 24/03/25	dom 06/04/25
TD Learning	14 días	lun 07/04/25	dom 20/04/25
Model Comparison	7 días	lun 21/04/25	dom 27/04/25
Deep Reinforcement Learning	**49 días**	**lun 28/04/25**	**dom 15/06/25**
Deep Q Learning	35 días	lun 28/04/25	dom 01/06/25
Model Comparison	14 días	lun 02/06/25	dom 15/06/25
Reports	**113 días**	**lun 10/03/25**	**lun 30/06/25**
Progress Report 1	35 días	lun 10/03/25	dom 13/04/25
Progress Report 2	42 días	lun 14/04/25	dom 25/05/25
Final Report	21 días	lun 26/05/25	dom 15/06/25
Final Delivery	15 días	lun 16/06/25	lun 30/06/25

Fig. 8: Gantt Diagram from Microsoft Project

A.2 Policies of 8x8 maps

↑	→	→	→	→	→	→	→
↑	↑	↑	↑	↑	↑	↑	→
←	←	←	None	→	↑	↑	→
←	←	←	↓, ↑	←	None	→	→
←	↑	←, ↑	None	→	↓	↑	→
←	None	None	↓, →	↑	←	None	→
←	None	↓, →	←, ↑	None	←, →	None	→
←	↓	←	None	↓, →	→	↓	G

Fig. 9: Value-Iteration Policy obtained in the 8x8 map

↑	↑	↑	→	→	→	→	→
↑	↑	↑	↑	→	→	→	→
↑	↑	←	None	→	↑	→	→
↑	↑	↑	↑	←	None	→	→
↑	↑	↑	None	→	↓	↑	→
↑	None	None	→	↑	←	None	→
↓	None	↑	↑	None	←	None	→
↑	None	←	None	→	→	↓	G

Fig. 10: Monte Carlo Policy obtained in the 8x8 map

↑	→	→	→	→	→	→	→
↑	↑	↑	↑	↑	→	→	↓
↑	↑	←	None	→	↑	→	↓
↑	↑	↑	↓	←	None	→	→
←	↑	↑	None	→	↓	↑	→
←	None	None	↓	↑	←	None	→
←	None	→	↑	None	→	None	→
←	↓	←	None	↓	↓	↓	G

Fig. 11: Q-Learning Policy obtained in the 8x8 map

A.3 Epsilon Decay in DQN

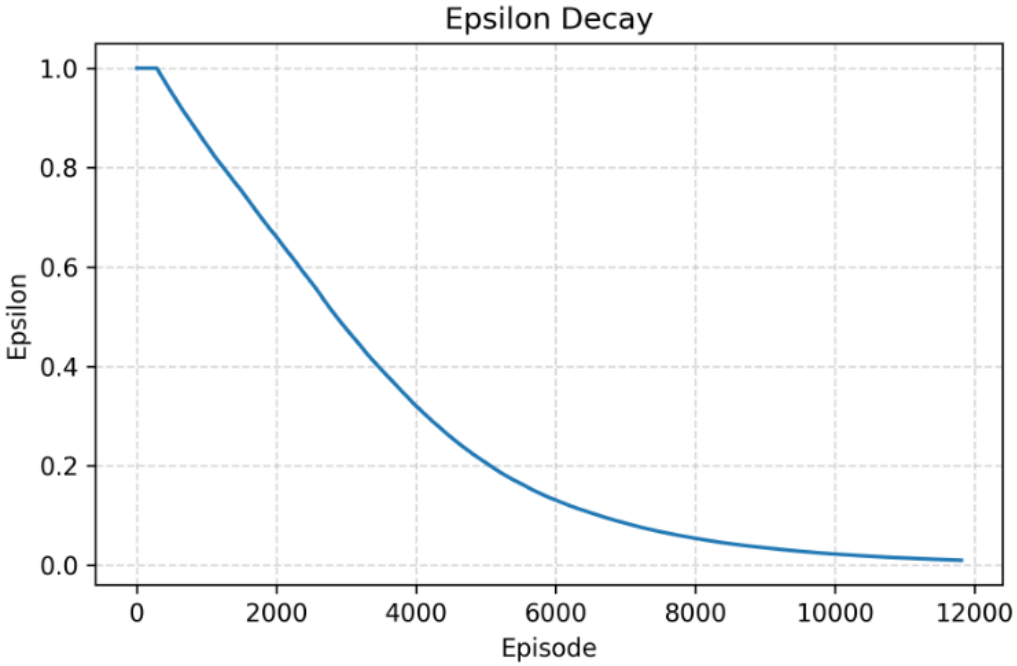


Fig. 12: Epsilon Decay Graph in DQN