Reinforcement Learning in Video Games

Valentí Torrents Vila

July 2, 2024

Resum– Aquest projecte explora l'aprenentatge reforçat (RL) en videojocs, una tècnica de machine learning que entrena un agent per aprendre a jugar a videojocs pel seu compte. Aquest projecte combina recerca teòrica amb implementacions pràctiques des de mètodes tabulars fins a mètodes basats en aprenentatge profund. Comença amb una introducció als fonaments de RL, aplicant solucions tabulars com Q-learning a l'entorn de Frozen Lake. A continuació, es dedica a resoldre el repte CartPole utilitzant approximate solution methods, i a millorar els resultats implementant una feedforward Deep Q-Network (DQN). El projecte culmina amb el desenvolupament d'una xarxa neuronal convolucional (CNN) DQN per abordar el joc Pong d'Atari. Els resultats emfatitzen l'adaptabilitat i el potencial de RL en videojocs, destacant millores significatives en la consistència d'aprenentatge i el rendiment a través d'arquitectures neuronals avançades.

Paraules clau– Aprenentatge Reforçat, Mètodes Tabulars, Programació Dinàmica, Mètodes Monte Carlo, Q-learning, Deep Q-Network, Convolutional Neural Networks, Function Approximation.

Abstract– This project explores reinforcement learning (RL) applications in video games, a machine learning technique that trains an agent to learn how to play video games on its own. Ranging from tabular methods to more advanced deep learning-based approaches, this project combines theoretical research with multiple practical implementations. It begins with an introduction to RL fundamentals, applying tabular solutions like Q-learning to the Frozen Lake environment. Then goes into solving the CartPole challenge using approximate solution methods, and improving those results implementing a feedforward Deep Q-Network (DQN). The project culminates with the development of a convolutional neural network (CNN) DQN to tackle the Atari Pong game. Results emphasize the adaptability and potential of RL in video gaming, highlighting significant improvements in learning consistency and performance through advanced neural architectures.

Keywords– Reinforcement Learning, Tabular Methods, Dynamic Programming, Monte Carlo Methods, Q-learning, Deep Q-Network, Convolutional Neural Networks, Function Approximation.

1 Introduction

1.1 Context

Reinforcement learning is a type of machine learning centered around how an intelligent agent should behave in an environment in order to achieve an specific goal. [3] Unlike other machine learning methods, like supervised learning or unsupervised learning, reinforcement learning's (RL) goal is to generate an intelligent agent that learns on its own while interacting with a dynamic environment. Through trial and error, the agent must chose which of the possible actions yield the most reward (both immediate and long-term), until achieving the necessary "knowledge" to go through the environment without any problems and solving it.

- E-mail: valentovi55@gmail.com
- Specialisation: Computation
- Work tutored by: Jordi Casas Roma
- Year 2023/2024

1.2 Objectives and expected results

The goal of this project is to dive into the world of Reinforcement Learning while using the Gymnasium library [2] effectively. Going from the fundamentals of RL, the project aims to understand and implement different RL algorithms. Starting with a quick introduction, and going through the basics with different tabular methods, our final objective is to go deep into the most complex methods applicable to video games. The following objectives mark the trajectory of this project:

1. Develop foundational understanding of Reinforcement Learning and proper use of the Gymnasium library.
2. Study and implement tabular methods such as Dynamic Programming, Monte Carlo, or Q-learning to create an agent capable of solving simple environments.
3. Study and comprehend some non-tabular methods, like policy-based methods and Deep Q-Networks (DQN) [7].
4. Apply the acquired knowledge to solve more complex and sophisticated environments, enhancing the agents' capabilities.
5. Generate a series of agents, each more complex, capable of achieving desired results in various games, while documenting findings and utilizing visualization tools to ensure a thorough exploration of Reinforcement Learning in video games.

1.3 Metodology

We have chosen to work using the Agile methodology for this project, working in iterative sprints each lasting two weeks. Using this methodology allows continual reassessment and adjustment if necessary since reinforcement learning is a dynamic field and it will need some adaptability. Through the regular review sessions with the tutor, we will check the gradual progression of the project, and provide checkpoints for evaluation, adaptation, and resolution of new obstacles as they appear.

1.4 Planning

The project planning is visualized through a Gantt Diagram in figure 9. This will be a dynamic tool to help the development of the project. At first it shows the main objectives and milestones necessary and, over time, using the Agile methodology, the Gantt diagram will show more, depending on the obstacles faced and the necessary changes that appear in the process.

2 STATE OF THE ART

Having gone through the basics of this project, we can now get into a more detailed explanation of what is the State of the art like, and how our first implementations of reinforcement learning methods has been like.

2.1 Introduction to RL

As we have just explained in the introduction, RL stands at an in-between point of artificial intelligence and decision-making. Thanks to that, it offers a powerful structure for creating and teaching agents to navigate through sequential decisions in dynamic environments. Simply put, RL revolves around the interaction between an agent and a specific environment. Through observations, this agent can perceive its environment and, based on its current state, selects one action or another. Furthermore, and depending on the specifics of the environment, the agent receives feedback in the form of rewards. Depending on the actions taken by the agent in each situation, these rewards can vary, indicating the desirability of said actions. Through these rewards, the agent is guided towards learning optimal strategies. To further understand how Reinforcement Learning works, we have to talk about its components:

- The environment is the problem space with which the agent interacts. All the states, possible actions, rewards and rules that the agent has to abide come from the environment. Its dynamics dictate how the agent's actions modify future states and rewards, and even though we usually see dynamic environments in RL, there can also be static ones.
- The agent is the principal entity in RL, and the one that takes the decisions, earns the rewards, and goes through the environment. By navigating through the environment, and recieving different inputs from it, the agent learns to make better decisions. Its goal is to follow a behaviour (policy) which maximizes the rewards over time, solving then the task at hand.
- Actions are the choices available to the agent at each step within the environment. Depending on which action the agent takes, its surroundings will be influenced one way or another. Actions can be discrete (finite options) or continuous.
- The states represent the current position or configuration of the environment. Depending on the moment and the position of the agent, the state changes and, with it, all the relevant information used by the agent to make decisions.
- Rewards are provided by the environment to the agent, and are signals that evaluate its actions. They can present inmediate or delayed feedback, and are the resposibles for the changes in the agent's behaviour over time, since its objective is to maximize these rewards in order to solve the environment.
- Policies define the agent's behavior by relating all states to actions. They mark the strategy followed by the agent to achieve its objectives. These policies can be deterministic, where each state has a specific action, or stochastic, where all actions have a probability of happening depending on the state and policy parameters. The objective of all Reinforcement Learning algorithms is to achieve the optimal policy v_π that maximizes long-term rewards, and leads the agent to solving the environment.

By interacting iteratively with its environment, and learning from its experience, RL algorithms generates agents that learn autonomously how to behave, solving different tasks, from video games to controlling robots. By learning through trial and error, and without supervision, RL ends up being perfectly adequate to solving scenarios where the agent must adapt to dynamic and complex environments.

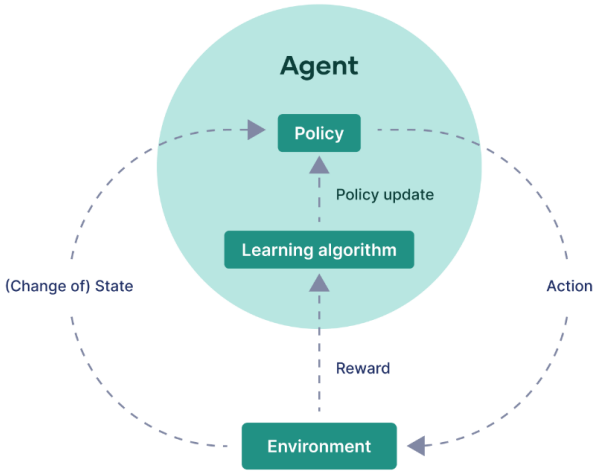


Fig. 1: RL General Framework [1]

2.2 Gymnasium

Gymnasium [2] serves as a versatile platform for the development and evaluation of reinforcement learning algorithms. Offering a diverse array of environments spanning from classic control problems to complex simulated scenarios, Gymnasium provides a standardized interface for interacting with different tasks. Each environment in Gymnasium is encapsulated as a Markov decision process (MDP) [3], allowing agents to interact with states, take actions, receive rewards and transition between states based on probabilistic dynamics. With its user-friendly API and extensive documentation, Gymnasium [2] empowers researchers and practitioners to prototype, benchmark, and iterate on various reinforcement learning techniques with ease. Moreover, Gymnasium's compatibility with popular RL libraries and frameworks fosters collaboration and accelerates the advancement of RL research and applications.

2.3 RL Methods

In the current state of reinforcement learning (RL), researchers employ tabular and non-tabular methods. Tabular methods, like dynamic programming and Monte Carlo, excel in small-scale tasks with discrete state and action spaces. They offer theoretical guarantees but struggle with larger, continuous spaces. Non-tabular methods, including deep reinforcement learning (DRL), leverage function approximation, particularly neural networks, to handle complex, high-dimensional environments. DRL algorithms like deep Q-networks (DQN) and policy gradients have demonstrated success in diverse applications, from robotics to gaming. Both approaches complement each other, with tabular methods providing theoretical foundations and non-tabular methods offering scalability and adaptability to real-world problems.

3 Tabular Methods

To get to understand the basics of RL we first have to talk about the tabular methods. These are the simplest ones in RL, since they store a specific action for each state in a table-like structure. The tabular methods we will talk about are Dynamic Programming, Monte Carlo methods, and Q-learning. To do so, we will work with finite Markov Decision Processes (MDP), trying to estimate value functions, which are functions of states that tell use how positive is for the agent to be in a specific state. [3]

3.1 Markov Decission Processes

A Markov Decission Process (MDP) is a mathematical model used in RL to make sequential decisions in a stochastic environment. Basically, an MDP describes a system (or environment) where an agent makes decisions which results are subject to uncertainty. Just like the parts of a RL environment, a MDP contains states, actions and rewards. However, a MDP also contains a Transition Model, which describes the transition probabilities from one state to another after taking a certain action. In order to solve a finite MDP, we can use different tabular methods, which we are going to talk about now.

3.2 Dynamic Programming

Dynamic Programming (DP) [3] is used to find the optimal policies to follow in order to solve a finite MDP. In a finite MDP, states, actions, and rewards are finite, and their dynamics are represented by transition probabilities. In order to achieve a good policy in RL in general, we use value functions. With DP we can compute these value functions by satisfying the Bellman optimality equations.
The Bellman Optimality Equations express the principle of optimality, that an optimal policy can be decomposed into smaller subproblems, each of which is solved optimally. In the field of MDPs, the Bellman Optimality Equations [3] are:

$$v_*(s) = \max_a \sum_{s',r} p(s', r \mid s, a)\,[r + \gamma v_*(s')] \qquad (1)$$

$$q_*(s,a) = \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right] \qquad (2)$$

In our project, we employed policy iteration to find this optimal value function. First, we initialized a policy, and then evaluated and improved it iteratively until it converged. To keep evaluating the policy we had, we estimated the value function with the current policy using the Bellman Equations. Afterwards, we selected the actions that maximized the expected returns in order to update the policy. After repeating this 2 steps several times and observing no further improvements, we had the optimized value function and policy that let us guide the decision-making through the MDP.

3.3 Monte Carlo Methods

Even though Monte Carlo and DP are both tabular methods, Monte Carlo methods do not require a model of the environment to operate. Instead, they rely on trial and error, sampling trajectories by interacting with the environment.
In our Monte Carlo policy evaluation process, we began by

executing episodes according to the current policy, collecting state-action-reward trajectories. After each episode, we calculated returns for each visited state, providing estimates of their values. These returns were then averaged over multiple episodes to update the value function incrementally. Optionally, we improved the policy based on the estimated values, possibly selecting actions greedily to maximize returns. This iterative refinement continued over multiple episodes until convergence, where the estimated value function approached the optimal one as the number of episodes increased.

3.4 Q-learning

Now that we have talked about both Monte Carlo and DP methods, we can get to understand temporal-difference (TD) learning [3], one of the most important methods of RL. TD is a mix of DP and Monte Carlo ideas, since it can learn directly from experience like Monte Carlo, and also update estimates from past estimates without having to wait for the final outcome, like DP.

From all the different TD methods, we just have worked with Off-Policy Q-learning [3], because it presents a different logic from the rest of the tabular methods. In Q-learning, the agent maintains an estimate of the value of each state-action pair represented by the action-value function Q(s, a). This function quantifies the expected cumulative reward that the agent will receive by taking action a in state s, and then following the optimal policy. With Q-learning, we aim to iteratively improve this estimate until it converges to the optimal action-value function. However, in off-policy Q-learning, we find that the agent learns from experiences generated by following a different policy than the one being learned.

Off-policy Q-learning allows agents to learn from past experiences generated under different policies, leading to faster learning. It effectively balances exploration (trying new actions) and exploitation (leveraging known information) in complex environments. This approach is versatile, enabling both policy evaluation and improvement, and benefits from experience replay to stabilize learning. Additionally, it's suitable for batch learning settings, where agents learn from fixed datasets, making it practical in scenarios where online exploration is challenging or costly.

3.5 Tabular Methods testing and results

To further understand the algorithms behind all three tabular methods we have talked about, we applied them to the Frozen Lake environment from the Gymnasium library [4]. This environment, as shown in figure 2, consists on helping an elf go from the initial state (top left) to the end state (bottom right). With the is_slippery parametter on True, this environment becomes slightly more complicated than it appears to be, making it non-deterministic. Without being slippery, the character would move where you tell it to. However, with the slippery on, the agent has a 33.3% chance of moving to the given direction, and 33.3% chance of moving to any of the two contiguous directions (e.g. if you choose right it can go right, up or down). The movements are: 0: LEFT, 1: DOWN, 2: RIGHT, 3: UP.

Fig. 2: Gymnasium Frozen Lake 4x4

With DP, we followed the pseudocode in algorithm 1 to achieve the optimal policy. The policy generated was $\pi = [0, 3, 3, 3, 0, 0, 0, 0, 3, 1, 0, 0, 0, 2, 1, 0]$, with a success rate of 82%. If we used the 8x8 Frozen Lake environment, this percentage went to a 100%, since there was a possible route with no chances of failing.

The algorithm used to implement Monte Carlo is the one shown in algorithm 2. The policy generated with MC was $\pi = [0, 3, 0, 3, 0, 0, 0, 0, 3, 1, 0, 0, 0, 2, 1, 0]$, which led the agent to solve the environment 78% of the time.

For Q-learning, the pseudocode was the algorithm 3. The policy generated was the same as in DP, making them equally efficient at solving the 4x4 Frozen Lake environment.

Algorithm 1: Policy Evaluation

Data: π, the policy to be evaluated,
A threshold $\theta > 0$ accuracy of estimation,
$V(s)$ initialized arbitrarily.
while $\Delta > \theta$ **do**
  $\Delta \leftarrow 0$
  foreach $s \in S$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow$
    $\sum_a \pi(a|s) \sum'_s, rp(s', r|s, a)[r + \gamma V(s')]$
    $\Delta \leftarrow max(\Delta, |v - V(s)|$

Algorithm 2: First-visit MC prediction

Data: π, the policy to be evaluated,
$V(s)$ initialized arbitrarily,
Returns(s) $\leftarrow$ an empty list.
while *forever* **do**
  Generate an episode following π:
  $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  for $t = T - 1$ **to** 0 **do**
    $G \leftarrow \gamma G + R_{t+1}$
    if S_t *not in* $S_0, S_1, \ldots, S_{t-1}$ **then**
      Append G to Returns(S_t)
      $V(S_t) \leftarrow$ average(Returns(S_t))

Algorithm 3: Q-learning Off-policy

Data: Step size $\alpha \in (0, 1]$,
$\epsilon > 0$
$Q(s, a)$ for all $s \in S^+, a \in A(s)$ initialized arbitrarily.
foreach *episode* **do**
  Initialize S
  foreach *step of episode* **do**
    Choose A from S using policy derived from Q(ϵ-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma max_a Q(S', a) - Q(S, A)]$
    $S \leftarrow S'$
  until S is terminal

4 APPROXIMATE SOLUTION METHODS

Now that we have understood the basics of RL, we can dig into a more complex field. Unlike when we were working with tabular methods, most of the state spaces we will be working with RL will be extremely complex and large. That is why we cannot aim to find an optimal policy v_π, but instead to find an approximate solution. To do so, the main tool we will be using is Generalization. Through generalizing the problems we face, we treat them like similar problems we have already faced, thus achieving proper results without the need of dedicating enormous quantities of time and data. In this second section of the project, we will be talking about different Approximate Solution Methods [3] and how to implement them in different environments.

4.1 On-Policy Prediction

We begin this chapter by focusing on the utility of function approximation to estimate the state-value function from on-policy data. Here, instead of representing v_π as a table, we use a parametrized functional form with a weight vector $w \in \mathbb{R}^d$. In practice, $\hat{v}(s, w)$ approximates $v^\pi(s)$, and this functional form might be realized within a neural network or as the criteria for split points in a decision tree.
It's important to remember that the dimensionality of w is generally much lower than the number of states. This means that updating the value of one state affects the values of others due to generalization, which offers more potent learning potential but also complicates management and understanding.
This chapter sets the foundation for understanding on-policy approximations in reinforcement learning, highlighting their theoretical underpinnings and practical implications in both fully and partially observable environments.

4.2 Value-function Approximation

Up until now, when we updated a specific state, we only shifted a tiny bit the overall behaviour of our agent towards the desirable policy, by modifying only the estimated value of the state we updated. Each of these updates is represented by $s \mapsto u$, where s is the state updated and u is the update target where s is shifting towards. Now, with generalization, the updates at s also change the estimated values of many other states. Machine learning methods that do this are called $supervisedlearningmethods$, and when the outputs are numbers, we call them $functionapproximation$. To generate an estimated value function using these methods, we take the $s \mapsto u$ of each update as training examples. By doing this, we enable these methods to be treated like any other learning method, and can now apply some of their algorithms. However, due to the necessity of having interactions and live updates, we need methods able to handle non stationary target functions.

4.3 PREDICTION OBJECTIVE

Through all this chapter, we have talked about the weight vector w but, how can we choose a proper w?
In tabular methods, the learned value function could end up being the true value function. However, here it cannot. Changing one state affects others, so we will never reach the correct value for all of them at the same time. That is why we need to give more importance to some of the states, specifying a distribution $\mu(s) \geq 0, \sum_s \mu(s) = 1$, which represents how much we care about the error in each state. The error here is calculated with the difference between the approximate value $\hat{v}(s, w)$ and the true value $v_\pi(s)$. Knowing all of this, we can now create an objective function, $\overline{VE}(w)$, which we will want to minimize in order to obtain the best w.

$$\overline{VE}(\mathbf{w}) \doteq \sum_{s \in \mathcal{S}} \mu(s) \left[v_\pi(s) - \hat{v}(s, \mathbf{w})\right]^2 \tag{3}$$

At first, this formula seems too complicated to apply, since we are summing through an exponentially big space (μ), and we are working with the true value v_π which we don't know. However, there are some algorithms with which we can approximately optimize it.

4.4 Stochastic Gradient Descent

In order to approximately optimize what we just talked about, we can use Stochastic Gradient Descent, also known as SGD. To work with SGD, we need to make several assumptions. Since we will be updating w as we bounce through the state space according to the on-policy distribution, all states must be visited in proportion to μ, which is what we have chosen from the on-policy distribution. Furthermore, our value function $\hat{v}(s, w)$ has to be differentiable with respect to w, necessary to calculate the gradient. And, as we said before, there needs to be a surrogate for $v_\pi(S_t)$, which is U_t.
With all this, SGD provides us with an update rule to apply to w for each visit to a state.

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)\right] \nabla \hat{v}(S_t, \mathbf{w}_t) \tag{4}$$

Where α is the step-size parameter, and $\nabla \hat{v}(S_t, w)$ is the gradient. Following this rule, SGD methods adjust w by a small ammount in the direction that would reduce most the error in that specific visit to that state.
Even though it may not seem obvious at first, it is necessary to carefully select which is the value of α. If the step on a certain direction was too big, we could end up with a value function with zero error in a certain state, but an abysmal

error in others. Taking small steps, and gradually adjusting w, will let SGD methods converge to a local optimum.
We now focus on the target U_t. If the expected value of U_t in a state S_t equals the true value $v_\pi(s)$, we call U_t unbiased, and w will undeniably converge into a local optimum of $\overline{VE}$.

4.4.1 Examples of Gradient Descend methods

Monte Carlo methods involve generating states by interacting with the environment using a policy π and updating the weights using the return G_t, an unbiased estimate of $v_\pi(S_t)$. The weight update using Monte Carlo is shown below:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[G_t - \hat{v}(S_t, \mathbf{w})\right] \nabla \hat{v}(S_t, \mathbf{w}) \tag{5}$$

However, Monte Carlo's reliance on complete episodes for updates slows down the learning process. An alternative is bootstrapping, which uses current estimates of future returns for more frequent updates. This approach introduces bias since U_t now depends on w_t, making it a semi-gradient step:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})\right] \nabla \hat{v}(S, \mathbf{w}) \tag{6}$$

Despite potential complications, semi-gradient methods, like semi-gradient TD(0), can offer faster convergence compared to Monte Carlo, similar to advantages observed with TD methods such as Q-learning.

4.5 Linear Methods

In function approximation, linear methods are crucial for the simplicity and effectiveness they bring. With these methods we approximate the state-value function as a linear combination of weights ($\mathbf{w}$) and feature vectors ($\mathbf{x}(s)$), represented by:

$$\hat{v}(s, \mathbf{w}) = \mathbf{w}^\top \mathbf{x}(s) \tag{7}$$

Following the algorithm used in the previous chapter, we can now see how this linear structure simplifies the update rule in Monte Carlo and temporal difference methods.
In Monte Carlo methods, the weight update formula is:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha(G_t - \hat{v}(S_t, \mathbf{w}))\mathbf{x}(S_t) \tag{8}$$

For gradient TD methods, the semi-gradient TD(0) update becomes:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})]\mathbf{x}(S) \tag{9}$$

These rules adjust the weights based on the temporal difference error, making the updates directly proportional to the prediction error, which enhances efficiency and makes it easier to interpret.
Feature construction also plays a crucial role in the application of linear methods. Utilizing polynomial features allows the model to capture complex interactions among state variables. For example, with two variables s_1 and s_2, the polynomial features are represented as:

$$x(s) = (1, s_1, s_2, s_1^2, s_1 s_2, s_2^2)^\top \tag{10}$$

These features allow the model to approximate more complex value functions.

4.6 Approximate solution methods testing and results

In this section, we present our experiments using the CartPole environment [5] from the Gymnasium library to evaluate the performance of Gradient Monte Carlo (GMC) and Gradient Temporal Difference (GTD) methods. The goal was to analyze the efficacy of these methods in reinforcement learning tasks, specifically in terms of their ability to balance the pole and maintain the cart within bounds.

4.6.1 Introduction to the CartPole Environment

The CartPole environment is a classic reinforcement learning problem where a cart, which can move horizontally along a track, has a pole attached to it. The state space consists of four continuous variables: cart position, cart velocity, pole angle θ, and pole angular velocity $\hat{\theta}$. The action space is discrete with two possible actions: pushing the cart left or right. We can see a representation of it in figure 3

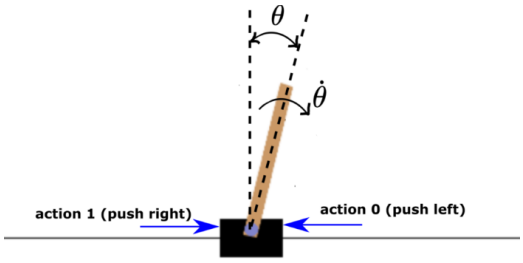


Fig. 3: Illustration of the CartPole environment [6]

The primary goal is to keep the pole balanced for as long as possible. The environment terminates when the pole falls past a certain angle or the cart moves too far from the center. The agent receives a reward for each time step the pole remains upright, encouraging strategies that maintain balance. Balancing the pole requires continuous and precise adjustments to the cart's position and velocity, making the CartPole environment an excellent testbed for evaluating reinforcement learning algorithms.

4.6.2 Results and Discussion

At first, we measured both methods in terms of their ability to maximize the total reward over a series of episodes. The results were surprisingly poor, leaving us questioning what was wrong. We tried implementing the algorithms using regular gradient descend, as well as applying lineal methods. By using polynomial features, we went from having the 4 main features to having 15, including the bias. After applying these changes, we got to see some improvement, but it was not enough.
In the GMC implementation, with a learning rate of 0.001, a discount factor of 0.95, a exploration rate ϵ of 0.01 and training for 100,000 episodes, the final weights of the features looked like this (see figure 4): As we can see, most of the feature weights converged to reasonable values. However, the bias weight always went to a number close to 8. This made the whole model shift to a higher value when deciding which action to take, rendering the model as a

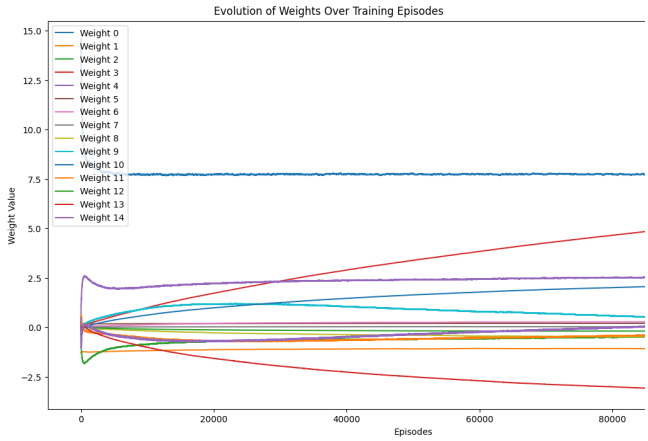


Fig. 4: Evolution of feature weights over time

failiure. However, by testing manually different values, we found that if we changed the bias' weight to a number close to 0.02 the model worked much better, going from an average reward per episode of 10 to an average of 150 steps until termination.
The Gradient Temporal Difference method, specifically using TD(0), updates the value function incrementally based on the temporal difference error. In the implementation of this method we tried different values for the parameters, and after seeing no kind of improvement, we decided to stick with the same we used in GMC. Although GTD was supposed to be faster at converging than GMC, the bias weight, as well as the others, remained a persistent issue, leading to less stable policy updates (see figure 5).

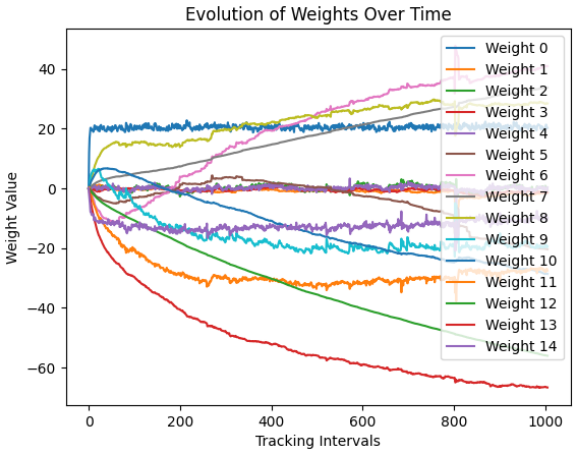


Fig. 5: Evolution of feature weights over time

After having trained the agent for 100,000 episodes, and not seeing a proper convergence, the average reward per episode was 15. Having reviewed the whole code, and finding no way of improving from the actual state, we decided to focus on the next section of the project in hopes of obtaining better results.
The implementations of SGD methods highlighted the challenges associated with the bias weight in both Gradient Monte Carlo and Gradient Temporal Difference. Even though both methods showed potential, addressing the bias weight issue is crucial if we want to achieve more proper and stable results.

5 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) combines reinforcement learning (RL) with deep learning [9]. In RL, an agent learns by interacting with an environment to maximize rewards. DRL uses deep neural networks to handle complex, high-dimensional environments wit which traditional RL would struggle.

5.1 About DRL

DRL excels in solving problems in large state spaces and complex environments, making it really useful for advanced applications. DRL can learn from raw data, enabling more sophisticated decision-making.
The main field where DRL is implemented are:

- **Video Games:** Achieving superhuman performance in complex games like Go, Chess, and Atari games.
- **Robotics:** Tasks like manipulation, locomotion, and navigation using sensory inputs.
- **Natural Language Processing:** Dialogue generation, machine translation, and text-based games.

5.2 Deep Q-Networks

Deep Q-Networks (DQNs) integrate Q-learning with deep neural networks to address high-dimensional state spaces. DQN employs a neural network to approximate the Q-function, predicting the maximum expected future reward for an action in a specific state.
The key components of the DQN architecture include:

- **Function Approximation with Neural Networks:** DQN uses a neural network to approximate Q-values, processing complex environments and potentially handling image inputs.
- **Experience Replay:** This mechanism breaks correlation between experiences by storing and randomly sampling them from a buffer, enhancing the training stability.
- **Fixed Target Network:** A stable copy of the Q-network, $Q(s, a; \theta^-)$, updates less frequently to avoid divergence in training, using:

$$y = r + \gamma \max_{a'} Q(s', a'; \theta^-) \tag{11}$$

- **Loss Function and Optimization:** DQN minimizes the mean squared error between predicted and target Q-values, updating network parameters θ via optimization methods:

$$L(\theta) = \mathbb{E}_{(s,a,r,s')\sim D}\left[\left(y - Q(s, a; \theta)\right)^2\right] \tag{12}$$

- **Exploration-Exploitation Trade-off:** DQN employs an ϵ-greedy policy, which decreases ϵ over time to transition from exploration to exploitation.
- **Handling High-Dimensional Inputs:** DQN utilizes convolutional neural networks (CNNs) for processing inputs like images, extracting relevant spatial features.

This configuration facilitates stable and efficient learning in DQNs by addressing key challenges of reinforcement learning in complex environments.

5.2.1 Types of DQNs

Having talked about DQNs, we now need to know how different deep neural networks can help create our DQN:

- **Feedforward Neural Network (FNN) DQNs:** They use a fully interconnected naural network to approximate the Q-value function. The network contains one input layer, multiple hidden layers with activation functions (like ReLu) and one output layer. The input is the state representation, and the output layer provides Q-values for each action. The network is trained using the Bellman equation to update Q-values based on rewards and future estimates. This type is simple to implement but struggles with high-dimensional inputs.
- **Convolutional Neural Network (CNN) DQNs:** These are ideal for processing high-dimensional state spaces like images. The architecture includes convolutional layers to detect spatial features, followed by fully connected layers. The input is typically an image or a stack of images, and the output layer provides Q-values for each action. Training involves updating Q-values based on rewards and future estimates using the Bellman equation.

As we just mentioned, Rectified Linear Units (ReLU) are an activation function used in both Feedforward Neural Networks (FNNs) and Convolutional Neural Networks (CNNs) to introduce non-linearity and help the networks learn more complex patterns. It outputs the input directly if it's positive, and zero otherwise. In FNNs, ReLU is applied after each hidden layer to improve training efficiency and model performance. In CNNs, it is used after each convolutional layer to activate features detected by filters, helping in the learning of intricate patterns.

6 DQN Testing and results

In this project, we implemented two different DQN to test. The first one is a DQN using feedforward neural networks to solve the cartpole environment (same one used in section 4.6). It is a more simple approach that allows us to understand the basics of a DQN, and lets us compare its results with the stochastic gradient implementations. The second one was a DQN using convulational neural networks to solve the pong environment from Gymnasium, a more complex environment that required a DQN capable of having image inputs. In both implementations we used the tensorflow library to generate the neural networks that will approximate the Q-functions in both DQNs.

6.1 Cartpole Environment using FNN DQN

The development process involves creating a Deep Q-Learning Network (DQN) to train an agent to balance a pole on a cart. The key steps include initializing the environment, designing the neural network, implementing the training loop, and managing the exploration-exploitation trade-off. The process is iterative, involving frequent evaluation and adjustment of key parameters. [8]

6.1.1 Initialization

First, we initialized the CartPole environment, providing state and reward information for the agent. Key parameters are defined for the DQN: the discount factor (gamma) which determines the importance of future rewards, typically set to 0.99; the exploration rate (epsilon) that controls the exploration-exploitation trade-off, initially set to 1, which decays over time; and the total number of episodes for training.

6.1.2 Neural Network Design

The DQN uses a feedforward neural network architecture. The input layer receives the state representation from the environment, which for CartPole is a vector representing position, velocity, angle, and angular velocity. The network is generated with two hidden layers with ReLU activation functions. The first hidden layer consists of 128 neurons, while the second hidden layer has 56 neurons. These neurons serve as the main computational units that process the inputs from previous layers and pass the output to the next. The output layer provides the Q-values for each possible action (move left or move right) and has two neurons with a linear activation function. The design also includes maintaining two networks: the main network and the target network. The main network is used to predict the Q-values for the current state, while the target network predicts the Q-values for the next state. The target network is updated periodically to stabilize the training process.

6.1.3 Replay Buffer

The replay buffer is initialized at the beginning of the code, and stores the experiences of the agent throughout training to allow it to learn from past interactions. Then, we randomly sample a batch of experiences from the replay buffer to train the neural network. By implementing a replay buffer and using batch sampling, the DQN is able to learn more efficiently and effectively, making better decisions over time as it is exposed to a wide variety of experiences.

6.1.4 Training Loop

For each episode, we reset the environment and initialized the variables to track rewards.
We then ran the episode until it terminated, or it got to 400 steps (to stop it from being too slow). In order to take a step, we first used the epsilon-gredy policy to select either a random action (exploration) or the one with the highest predicted Q-value (exploitation). After performing the action, we observed the next state, reward, and if it terminated. Finally, after storing the experience in the replay buffer, if it had enough experiences we sampled a batch and trained the main network, adjusting the target network periodically.

6.1.5 Results analysis

After training the agent for over 500 episodes, it got to a point where he got to the maximum possible number of

steps before terminating, thus balancing the cartpole properly.
We can see in figure 6 that there is a steady increase in the Q-value. This indicates that, as training goes on, the agent is learning to predict higher rewards, showing how the policy is improving the performance. Fluctuations may indicate exploration phases or instability in learning.

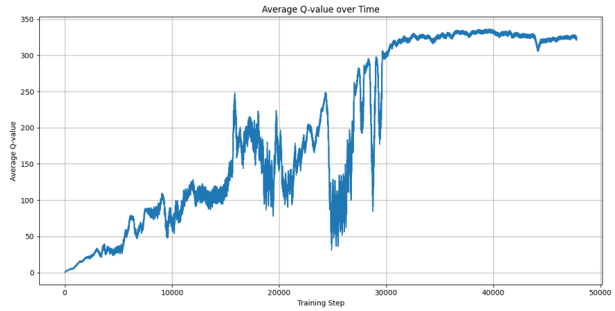


Fig. 6: Average Q-value over Time

In figure 10 we can observe that as training progresses ϵ decays, and the agent explores less and exploits more, relying on learned knowledge to make decisions.
We observe in figure 7 that, even though the agent kept getting better, there were still some fluctuations on the total rewards until the very end. This happened because even if the policy followed was already good, some possible initial states led the agent to fail. After more training, though, the agent corrected this and achieved a perfected policy.

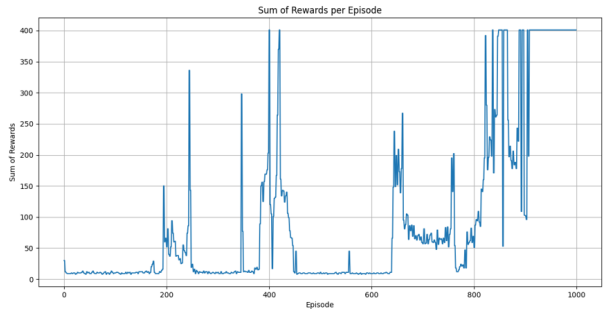


Fig. 7: Sum of Rewards per Episode

Overall, the results were really good. We tested different values for the discount factor, the exploration rate, and the total number of episodes to train for, and decided to work with the ones in the code.
For more information about the results, see cartpole_DQN's stored videos on the github [see appendix]

6.2 Pong Environment Using CNN DQN

The development of this part involved creating a DQN using convulational neural networks [9] to solve the Pong environment from Gymnasium [10]. Even though the basis of this implementation was the same as in the first one, having to use CNN to compute the image inputs made a huge difference.

6.2.1 Environment Description

The Pong environment from Gymnasium emulates a simplified table tennis game with one paddle at each horizontal end of the screen. The agent controls the right paddle, and competes against the left paddle controlled by the computer. Each one tries to keep bouncing the ball away from their goal and into their opponent's goal. The observable space for the agent is a matrix of pixel values and can choose from six discrete actions, these being to move the paddle up (both 2 and 4), down (3 and 5), or keeping it stationary (0 and 1). The objective is to maximize the score difference against the opponent, with the agent receiving a reward for each point scored and a penalty for points lost.

6.2.2 CNN Architecture

To effectively process the visual input from the Pong environment, a CNN is utilized. The CNN is created by 3 convulational layers, followed by a flatten layer, 2 fully connected layers and ending in the output layer. The input layer initiates the feature extraction by capturing basic spatial features such as edges and general shapes from the input images. Then, the intermediate CNN layers extract complex spatial features and provide detailed and localized analysis useful for decision-making. The flattening layer converts the 3D output from the convolutional layers into a 1D vector to make the fully connected processing easier. The dense layers then process the flattened vector, integrating and interpreting the extracted features, beginning the decision-making process. And finally, the output layer outputs the Q-values for each possible action based on the current state.
A more visual description of our CNN can be seen with keras' model.summary [11] as shown in figure 11.

6.2.3 Deep Q-Learning Framework

The different functions used on this implementation doesn't differ much from the previous one. We have the experience replay, which now samples batches of 32 experiences to use for training. We also have the epsilon-greedy policy, which works the same way as in the FNN DQN, having ϵ decay from 1 to 0.01 over time. We use functions like make_env, play_one_step and training_step, which allow a more structured functionallity of the code, simplifying actions like creating and modifying the pong environment, deciding which action to take, storing the experiences in the replay buffer, and other necessary steps for the proper implementation of the neural network.
Lastly, we used a training loop, which put everything together to train the agent, storing all the relevant data to allow data visualization.

6.2.4 Training and Evaluation

We trained the agent for 700 episodes. After having ϵ decay to 0.01, and seeing how the learned policy guided the agent to play pong, we decided to stop the training process. What we observed, unfortunately, was that the policy learned didn't lead the agent to behave properly on the environment, ending up with the worst total reward possible every single time. In figure 8 we can observe how, after relaying almost entirely on the Q-values, and not doing many random actions, the rewards flatlined. We consider that this implementation was well constructed and organized and, even though it concluded without proper results,

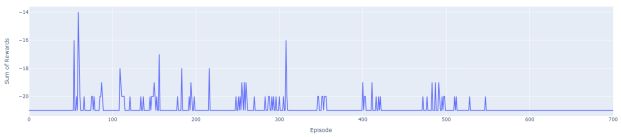


Fig. 8: Total sum of rewards per episode

it has been useful to get to understand more how a DQN using CNN works. With a different approach and more time, we hopefully would have obtained better results. Unfortunately, it was not possible due to the extension limitations of the project.

7 Conclusions

This project has demonstrated the adaptability and robust potential of RL techniques across different video game environments, through the application of tabular methods in simpler environments like Frozen Lake and the use of DQNs and Approximate Solution Methods to solve more complex environments like CartPole and Atari Pong.
RL has been demonstrated to consistently improve the efficiency and effectiveness of game playing agents. The implementation of a feedforward DQN in the CartPole environment, for example, showed the capacity of DRL to achieve a stable performance through refining the agent's responses even after already having proper results. Moreover, the CNN DQN developed for the Pong environment ended up being even more useful, being able to handle high-dimensional state spaces with a degree of sophistication that traditional methods could not achieve.
However, we cannot say the journey has been without difficulties. Even after understanding how different RL algorithms work, and beginning to implement them to solve different environments, there were many challenges to overcome. Both algorithm selection and parameter tuning were crucial to the proper developement of the algorithms, since a careless approach would mean inaccurate results. Each method, from dynamic programming to Monte Carlo and temporal-difference learning, to Function Approximation and DQns, presented unique benefits and limitations, reflecting the complexity present in RL applications.
Looking ahead, this project presents a solid basis for future investigations and implementations surrounding the potential scalability of RL methods, both in gaming, and in other areas. Next steps could revolve around searching for more optimal implementations to the ones we had trouble with, as well as integrating some of the already working RL implementations to environments not shown on this project.
The remarkable adaptability shown by RL methods to engage in a wide variety of challenges in video gaming not only emphasizes their potential within this area, but also illustrates the fact that a broader implementation of RL methods in technology and AI research would be prosperous. The insights and knowledge gained from this project can't help but encourage more thorough and deeper exploration in this continually evolving field.

Special Thanks

I want to thank my tutor Jordi Casas for letting me work in a project as interesting as RL, and helping me throughout the process. I also want thank my friends who listened to me ramble on and on about how was the project going. And thanks to my family that helped me get where I am now, and have been by my side since the beginning.

References

[1] The General Framework of Reinforcement Learning. (2023). [Online]. Available: https://www.scribbr.com/wp-content/uploads/2023/08/the-general-framework-of-reinforcement-learning.webp (Accessed: February 14, 2024)

[2] Gymnasium Documentation. [Online]. Available: https://gymnasium.farama.org/ (Accessed: February 17, 2024)

[3] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press, 2020.

[4] Gymnasium Frozen Lake Documentation. [Online]. Available: https://gymnasium.farama.org/environments/atari/pong/ (Accessed: February 28, 2024)

[5] Gymnasium Cartpole Documentation. [Online]. Available: https://gymnasium.farama.org/environments/atari/pong/ (Accessed: May 1, 2024)

[6] A. Haber. Cart Pole Control Environment in OpenAI Gym. [Online]. Available: https://aleksandarhaber.com/cart-pole-control-environment-in-openai-gym-gymnasium-introduction-to-openai-gym/ (Accessed: May 15, 2024)

[7] Jesse Farebrother, Marlos C. Machado, and Michael Bowling. Generalization and Regularization in DQN. [Online]. Available: https://doi.org/10.48550/arXiv.1810.00123 (Accessed: June 1, 2024)

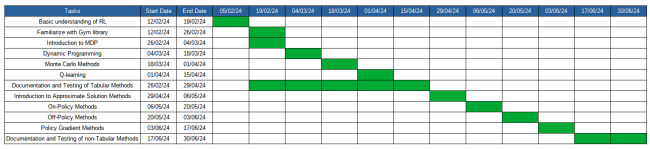
[8] A. Haber. Deep Q-Networks (DQN) in Python from Scratch by Using OpenAI Gym and TensorFlow: Reinforcement Learning Tutorial. [Online]. Available: https://aleksandarhaber.com/deep-q-networks-dqn-in-python-from-scratch-by-using-openai-gym-and-tensorflow-reinforcement-learning-tutorial/ (Accessed: June 10, 2024)

[9] Yuxi Li. Deep Reinforcement Learning: An Overview. [Online]. Available: https://arxiv.org/abs/1701.07274 (Accessed: May 22, 2024)

[10] Gymnasium Pong Documentation. [Online]. Available: https://gymnasium.farama.org/environments/atari/pong/ (Accessed: June 10, 2024)

[11] TensorFlow Keras Library Documentation. [Online]. Available: https://www.tensorflow.org/guide/keras (Accessed: June 10, 2024)

APPENDIX



Tasks	Start Date	End Date	05/02/24	19/02/24	04/03/24	18/03/24	01/04/24	15/04/24	29/04/24	06/05/24	20/05/24	03/06/24	17/06/24	30/06/24
Basic understanding of RL	12/02/24	19/02/24												
Familiarize with Gym library	12/02/24	26/02/24												
Introduction to MDP	26/02/24	04/03/24												
Dynamic Programming	04/03/24	18/03/24												
Monte Carlo Methods	18/03/24	01/04/24												
Q-learning	01/04/24	15/04/24												
Documentation and Testing of Tabular Methods	26/02/24	29/04/24												
Introduction to Approximate Solution Methods	29/04/24	06/05/24												
On-Policy Methods	06/05/24	20/05/24												
Off-Policy Methods	20/05/24	03/06/24												
Policy Gradient Methods	03/06/24	17/06/24												
Documentation and Testing of non-Tabular Methods	17/06/24	30/06/24												

Fig. 9: Gantt Diagram

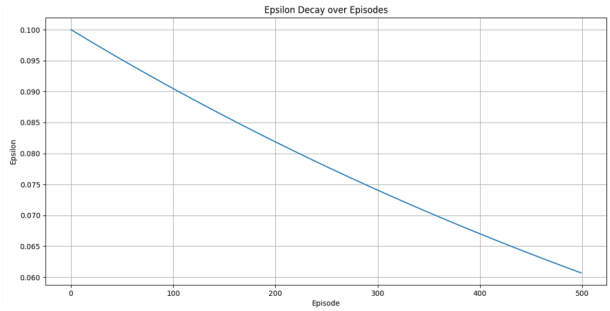
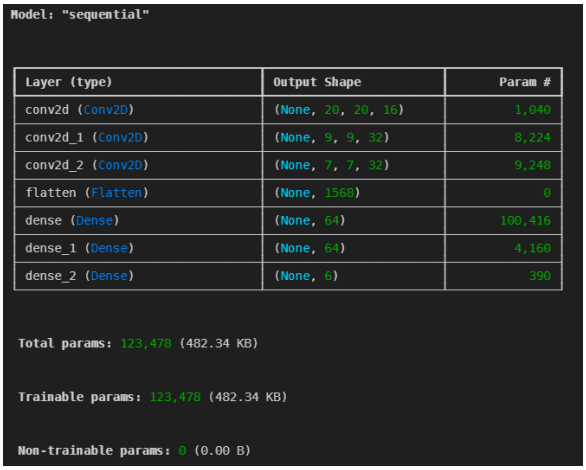


Fig. 10: Epsilon Decay over Episodes



Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 20, 20, 16)	1,040
conv2d_1 (Conv2D)	(None, 9, 9, 32)	8,224
conv2d_2 (Conv2D)	(None, 7, 7, 32)	9,248
flatten (Flatten)	(None, 1568)	0
dense (Dense)	(None, 64)	100,416
dense_1 (Dense)	(None, 64)	4,160
dense_2 (Dense)	(None, 6)	390

Total params: 123,478 (482.34 KB)

Trainable params: 123,478 (482.34 KB)

Non-trainable params: 0 (0.00 B)

Fig. 11: CNN model summary using Tensorflow Keras

Here is the link to the GitHub repository where all the code developed during this project is stored. URL to the repository: https://github.com/ValentiTorrents/TFG_Reinforcement_Learning