UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS

Maurício Garcia Di Mase

Controle via Aprendizado por Reforço
Control via Reinforcement Learning

São Carlos

2025

Maurício Garcia Di Mase

Controle via Aprendizado por Reforço
Control via Reinforcement Learning

Monografia apresentada ao Curso de Engenharia Elétrica com Ênfase em Eletrônica, da Escola de Engenharia de São Carlos da Universidade de São Paulo, como parte dos requisitos para obtenção do título de Engenheiro Eletricista.

Advisor: Prof. Dr. Marcos Rogério Fernandes

São Carlos

2025

Ficha catalográfica elaborada pela Biblioteca Prof. Sérgio Rodrigues Fontes
e pelo Serviço de Comunicação e Marketing da EESC-USP,
com dados inseridos pelo(a) autor(a).

G216c

Garcia Di Mase, Maurício

Controle via Aprendizado por Reforço / Maurício Garcia Di Mase ; orientador Marcos Rogério Fernandes. -- São Carlos, 2025.

70 p.

Monografia - Graduação em Engenharia Elétrica com ênfase em Eletrônica -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2025.

1. Aprendizado por Reforço. 2. Engenharia de Controle. 3. Aprendizado de Máquina. 4. Q-learning. 5. Actor-Critic. I. Fernandes, Marcos Rogério, orient. II. Título.

Responsáveis pela estrutura de catalogação da publicação segundo a AACR2: Bibliotecários da EESC/USP.

FOLHA DE APROVAÇÃO

Nome: Maurício Garcia Di Mase

Título: "Control via Reinforcement Learning"

Trabalho de Conclusão de Curso defendido e aprovado em 01/12/2025,

com NOTA 10,0 (dez , zero), pela Comissão Julgadora:

Prof. Dr. Marcos Rogério Fernandes - Orientador - SEL/EESC/USP

Prof. Dr. Eduardo Fontoura Costa - Professor Associado ICMC/USP

Prof. Dr. Ricardo Augusto Souza Fernandes - SEL/EESC/USP

Coordenador da CoC-Engenharia Elétrica - EESC/USP:
Professor Associado José Carlos de Melo Vieira Júnior

A Deus Pai todo poderoso, criador do céu e da terra
de todas as coisas visíveis e invisíveis.

ACKNOWLEDGEMENTS

A realização deste trabalho só foi possível graças ao apoio de diversas pessoas.

Agradeço, em primeiro lugar, ao meu professor orientador, Dr. Marcos Rogério Fernandes, por me ter introduzido no assunto e por me guiar com paciência e empenho.

Agradeço ternamente aos meus pais, que muito insistiram para que eu cursasse engenharia, disciplina que aprendi a amar.

Agradeço também a todos os meus professores, que contribuíram para minha formação acadêmica, e aos colegas que tornaram todo o processo mais agradável e frutífero.

Finalmente, reservo um agradecimento ao Centro Cultural Boa Vista, pela formação moral que tanto contribuiu para que eu pudesse levar este trabalho a cabo com diligência.

„Da steh' ich nun, ich armer Tor!
Und bin so klug als wie zuvor.“
Johann Wolfgang von Goethe

ABSTRACT

DI MASE, M. G. **Control via Reinforcement Learning**. 2025. 68 p. Monograph (Conclusion Course Paper) - Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2025.

This work presents a comprehensive review and practical application of Reinforcement Learning (RL) algorithms in control engineering. The theoretical groundwork of RL is laid out, establishing its connection to Optimal Control and detailing various algorithms, including Dynamic Programming (Value Iteration and Policy Iteration), Q-learning (Tabular and Deep Q-Learning/DQN), and Actor-Critic methods (Deep Deterministic Policy Gradient/DDPG and Twin Delayed Deep Deterministic Policy Gradient/TD3).

The algorithms are first validated by comparing Deep Q-Learning against Dynamic Programming for a simple discrete Markov Decision Process (MDP) with a small state space, demonstrating the capability of approximation methods to converge toward the exact optimal policy, although this is not guaranteed in larger environments. Subsequently, performance comparisons are conducted between the RL agents (DQN, DDPG, TD3) and a Linear Quadratic Regulator (LQR) in simulated environments for classic control systems: the simple pendulum, cart-pole, and rotary pendulum. Results show that while the LQR is highly effective near the unstable equilibrium point, RL agents, particularly TD3, demonstrate superior generality for initial conditions farther from the linearization point. The study also examines learning stability, confirming TD3's robustness against Q-value overestimation, a problem observed in DDPG and DQN training.

Finally, a novel hybrid control scheme combining TD3 for non-linear tasks (swing-up) and LQR for stabilization is proposed and implemented. This hybrid approach demonstrates a sensible reduction in total cumulative cost, proving particularly effective in the more complex, non-linear rotary pendulum system. The findings validate RL's relevance and potential as a robust alternative for designing controllers for complex, real-world non-linear systems.

Keywords: Reinforcement Learning. Control Engineering. Machine Learning. Q-learning. Actor-Critic. DQN. TD3. DDPG.

RESUMO

DI MASE, M. G. **Controle via Aprendizado por Reforço Aplicado**. 2025. 68 p. Monografia (Trabalho de Conclusão de Curso) - Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2025.

Este trabalho apresenta uma revisão abrangente e uma aplicação prática de algoritmos de Aprendizado por Reforço (Reinforcement Learning – RL) no domínio da engenharia de controle. A base teórica do RL é estabelecida, demonstrando sua conexão com o Controle Ótimo e detalhando diversos algoritmos, incluindo Programação Dinâmica (Iteração de Valor e Iteração de Política), Q-learning (Tabular e Deep Q-Learning/DQN) e métodos Ator-Crítico (Deep Deterministic Policy Gradient/DDPG e Twin Delayed Deep Deterministic Policy Gradient/TD3).

Os algoritmos são inicialmente validados por meio da comparação entre o Deep Q-Learning e a Programação Dinâmica em um Processo de Decisão de Markov (MDP) discreto e simples, com um pequeno espaço de estados, o que demonstra a capacidade dos métodos de aproximar a política ótima exata, embora isso não seja algo garantido para ambientes maiores. Em seguida, são realizadas comparações de desempenho entre os agentes de RL (DQN, DDPG, TD3) e um Regulador Linear Quadrático (LQR) em ambientes simulados de sistemas clássicos de controle: o pêndulo simples, o pêndulo invertido em carrinho (cart-pole) e o pêndulo rotativo.

Os resultados mostram que, embora o LQR seja altamente eficaz próximo ao ponto de equilíbrio instável, os agentes de RL, particularmente o TD3, demonstram maior generalidade para condições iniciais distantes do ponto de linearização. O estudo também examina questões relacionadas à estabilidade do aprendizado, confirmando a robustez do TD3 em relação à superestimação dos valores-Q, um problema observado durante o treinamento de DDPG e DQN.

Por fim, propõe-se e implementa-se uma nova estratégia de controle híbrido, combinando o TD3 para tarefas não lineares (como o balanço ou o swing-up) e o LQR para a estabilização. Essa abordagem híbrida demonstra uma redução significativa no custo cumulativo total, mostrando-se particularmente eficaz no sistema mais complexo e não linear do pêndulo rotativo. Os resultados obtidos validam a relevância e o potencial do RL como alternativa robusta para o projeto de controladores aplicados a sistemas não lineares complexos do mundo real.

Palavras-chave: Aprendizado por Reforço. Engenharia de Controle. Aprendizado de Máquina. Q-learning. Actor-Critic. DQN. TD3. DDPG.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AC	Actor-Critic.
DDPG	Deep Deterministic Policy Gradient.
DP	Dynamic Programming (Programação Dinâmica).
DQN	Deep Q-Learning ou Deep Q-Network.
DRL	Deep Reinforcement Learning.
HJB	Hamilton-Jacobi-Bellman (implícito em Optimal Control & HJB).
LQE	Linear Quadratic Estimator.
LQG	Linear Quadratic Gaussian.
LQR	Linear Quadratic Regulator (Regulador Linear Quadrático).
LTI	Linear Time-Invariant.
MDP	Markov Decision Process (Processo de Decisão de Markov).
MPC	Model Predictive Control (implícito em Deep MPC).
PID	Proportional-Integral-Derivative.
PPO	Proximal Policy Optimization.
RK4	Classic Runge-Kutta Method.
RL	Reinforcement Learning (Aprendizado por Reforço).
SAC	Soft Actor-Critic.
TD	Temporal Difference.
TD3	Twin Delayed Deep Deterministic Policy Gradient.
USP	University of São Paulo.
VIQL	Value-Iteration-based Q-learning.

LIST OF SYMBOLS

I	Moment of inertia.
J	Cost function in optimal control, $J_{0\to N}$..
M	Mass of the cart; also point mass on the pendulum tip.
P	State transition probability function, $P(s, a, s')$; also a weighting matrix in LQR cost J.
Q	Action-value function or Q-function, $Q(s, a)$.
R	Reward signal.
T	Kinetic energy.
V	State-value function, $V(s)$; also potential energy.
α	Learning rate (Tabular Q-Learning); also a shorthand variable in rotary pendulum dynamics.
β	Viscous friction coefficient; also a shorthand variable in rotary pendulum dynamics.
$\dot{\phi}$	Angular velocity (time derivative of ϕ).
$\dot{\theta}$	Angular velocity (time derivative of θ).
$\dot{x}$	Velocity of the cart (time derivative of x).
ϵ	Small value (convergence threshold); also used for exploration (ϵ-greedy) or noise ($\epsilon \sim \mathcal{N}$).
γ	Discount factor, $0 < \gamma \leq 1$.
$\in$	Element of / Belongs to.
$\mathcal{A}$	Action space (set of all possible actions).
$\mathcal{L}$	Lagrangian of the system, $\mathcal{L} = T - V$.
$\mathcal{S}$	State space (set of all states).
ϕ	Angle of the horizontal arm (Rotary Pendulum); also a potential function in reward shaping.
π	Policy (control rule); often $\pi(s)$ or $\pi(s, a)$.
τ	Torque control signal (u); also the target-network soft-update rate, $0 < \tau < 1$.
θ	Angle of the pendulum (vertical arm); also used for neural network weights/parameters.
a	Action chosen by the agent (control signal).
l	Length of the pole/pendulum (e.g., l_p, l_a).
m	Mass of the pole/pendulum (e.g., m_p, m_a).
s	State of the system (current state).
u	Control signal (general notation).

x	Position of the cart (Cart Pole); also a general state variable.
y	Target Q-value used in loss computation.

CONTENTS

1 INTRODUCTION

Reinforcement Learning is a set of tools for solving problems involving search, trial and error, and successive updates of the agent doing the searching, trying, and making mistakes (hence, "Learning") in order to maximize some sort of reward signal. That is a very general description, and indeed this subject has a ranging from robotics to gaming, finance, natural language processing, and of course control engineering. Sequential Decision Making applications (training a model to play games) in particular gained a lot of attention with the publication of the article "Human-level control through deep reinforcement learning" [Mnih *et al.* 2015] in which the authors trained Deep Q-Networks (DQN) to play classic Atari games using only raw pixel data as input; and another milestone for RL was the introduction of AlphaGo [Silver *et al.* 2016], the first artificial intelligence to beat the world champion of the game Go in a match.

Control applications have advanced significantly, with notable developments in recent years. A model-free optimal tracking design for nonlinear systems has been proposed, which learns control policies directly from measured data using a value-iteration-based Q-learning (VIQL) scheme [Wang, Huang e Zhao 2024]. In robotics, a bionic learning algorithm for a two-wheeled robot combines a growing cell structure network with Q-learning for balance control in a continuous state system [Hongge *et al.* 2014]. Another approach to a classic control problem is the swing-up control of a double inverted pendulum, achieved by combining Q-learning algorithms with a classical PID control scheme [Zeynivand e Moodi 2022]. The broader fields of robotic manipulation and fixed-wing aircraft control have also seen significant progress. A review of RL for fixed-wing aircraft control examines lvarious tasks such as attitude control, landing, and flocking, noting the widespread use of algorithms like PPO, DDPG, and SAC [Richter, Calix e Kim 2024].

Despite the many other applications of these tools, the mathematical foundations of the subject are deeply connected to Optimal Control. In Optimal Control, the controller is designed offline by solving the Bellman's equation associated with the system thereby minimizing a given cost function. The solution is computed through dynamic programming, the simplest form of which consists of making an exhaustive computation backward in time. For Linear Time-Invariant systems, Bellman's equation can be solved by solving Riccati's equation, which leads to a closed-form expression for the control rule. However, for non-LTI systems, such as an inverted pendulum on a cart, a dc motor with friction or a quadcopter drone, this is not the case, and dynamic programming could be usedneed to be employed in order to get an exact solution, but it would need to explore the state space of these systems exhaustively, since their dynamics cannot be described in terms of linear control theory.

The problem with this is that the iterative dynamic programming algorithm has a very high computational cost, suffering from what Bellman called the "curse of dimensionality", i.e. DP's computational requirement grows exponentially with the size of the state space [Sutton e Barto 2018]. Reinforcement Leaning tools offer a way to approximate the optimal policy (the equivalent of 'control rule' in Control Engineering terminology) instead of actually finding the exact optimal policy by solving Bellman's equation through DP.

Many such tools have been devised, e.g. tabular Q-Learning, Deep Q-Learning, Actor Critic Algorithms among many others, and the present work seeks to explore some of them, first comparing it with Dynamic Programming for a system with a small discrete state space, then applying it to different plants inside a simulation environment. One thing that was realized during the first semester of research is that there are many different ways to define the reward signal in the simulation environment, and nowadays, that is more of an art than a science. One of the goals of this research is to experiment with different approaches and report what worked best.

1.1 Problem Statement, Relevance and Objectives

The main objectives of this – – such as Deep Q-Networks, Deep Deterministic Policy Gradient and Twin Delayed Deep Deterministic Policy Gradientpresent work are to review Reinforcement Learning theory by breaking down different RL algorithms, namely Dynamic Programming, Q-learning, Deterministic Policy Gradient, and Soft Actor Critic, as well as the Neural Network based implementations of them. Then, some important observations are made about how to choose a reward signal based on seminal papers on the subject, that touch on policy invariant reward transformations and reward hacking, which were observed in practice in unexpected ways.

Once the theoretical remarks have been laid out, the algorithms are implemented and tested in simulated control system environments, and then the performance of each algorithm is compared against each other and against traditional control methods. The algorithms can only be compared with conventional control methods, such as a linear quadratic regulator in the linear regime, e.g., an inverted pendulum initialized near the upright position. It is expected that the farther the initial conditions are from the point about which the linearization was made to design the controller, the greater the advantage of RL agents over traditional controllers will be. Performance will be measured number of time-stepsthrough the total reward obtained by each agent in a set period of run .

Another goal of this work is to train RL agents to perform nonlinear control tasks, namely the swing-up task in the simple pendulum, cart-pole, and rotary inverted pendulum.

This work's relevance is multifaceted, offering a comprehensive review and practical application of the main Reinforcement Learning algorithms–from Dynamic Programming and Q-learning to DDPG and TD3 with the latter two being very relevant Critic, considered to be the state of the art — within the domain of control engineering applications. By implementing and comparing these advanced RL techniques against each other and against well-established traditional control methods, like the Linear Quadratic Regulator, the study provides empirical evidence of RL's potential, particularly demonstrating its superiority in non-linear and challenging control tasks (like and in regimes far from the linearization point. Furthermore, the practical investigation into reward signal design and the observation of reward hacking provide novel insights for the practice of RL. Future efforts will involve successfully demonstrating the application of trained RL agents on a physical system (the rotary inverted pendulum plant). This is expected to validate RL as a robust and promising alternative for designing controllers for complex, real-world non-linear systems, ultimately bridging the gap between theoretical RL advancements and practical control system implementation.

1.2 Organization of the Work

The work is divided into four main chapters: Methodology, Computational Experiments and Conclusions. Methodology is subdivided into several sections covering Optimal Control and Reinforcement Learning Theory, RL algorithms, how a reward signal is chosen in order to avoid reward hacking and work well in control engineering applications, and finally a section dedicated to describing the three systems that were tested: their dynamics, reward signal and simulated environments. Computational Experiments is divided into a section in which dynamic programming is compared against Deep Q-Learning for a simple RL problem with small dimensions, in order to show that Q-learning is capable of approximating the exact optimal value function found by DP; the second section in this chapter deals with the computational experiments done in order to compare the total reward gained with RL algorithms and with LQR; and the final third section compares the performance of each algorithm for non-linear control tasks. The final chapter deals with the main takeaways regarding the results in learning stability, performance evaluation and future work that will build upon what has been done.

2 METHODOLOGY

2.1 Optimal Control and Bellman's Equation

Optimal control is a branch of control theory based on an extension of variational calculus that seeks to define the sequence of control signals for a given system in a way that minimizes a cost function established by the problem (for example, equation 2.1 in the LQR problem). The system itself is defined by a dynamics function F, which takes the current state s_k and chosen action a_k and outputs the next state s_{k+1}, as in equation 2.2. Within this paradigm, various approaches have been proposed for both control and state prediction, such as the Linear Quadratic Regulator (LQR), the Kalman filter (LQE), and their combination, the Linear Quadratic Gaussian (LQG) controller.

$$J_{0\to N} = s_N^T P_N s_N + \sum_{k=0}^{N-1} s_k^T Q s_k + a_k^T R a_k. \tag{2.1}$$

$$s_{k+1} = F(s_k, a_k). \tag{2.2}$$

The general method for solving these problems boils down to solving the Bellman equation. Equation 2.3 is an example that emerges from the optimization problem of equation 2.1.

$$V_k(s_k) = \min_{a_k}[s_k^T Q s_k + a_k^T R a_k + V_{k+1}(s_{k+1})]. \tag{2.3}$$

A possible interpretation of the Bellman equation is that it represents the minimum cost to reach state s_N from state s_k. Because it is the minimum cost, it depends only on the cost of the current stage ($s_k^T Q s_k + a_k^T R a_k$) and the minimum cost from the next stage onward ($V_{k+1}(s_{k+1})$), also known as the cost-to-go.

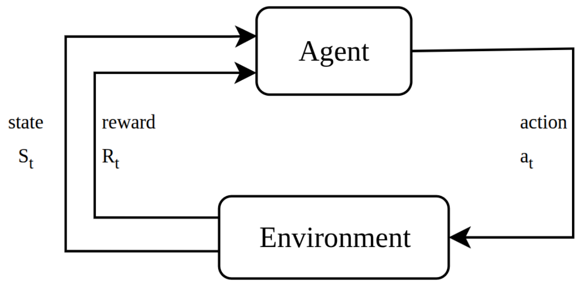
For linear time-invariant systems, this has a closed-form solution: the optimal state-feedback control gain, known as the LQR. One could, for example, use an LQR to control an inverted pendulum near its unstable equilibrium. Still, this solution alone is insufficient for implementing a control law for the pendulum's swing-up task. In other words, it cannot determine the sequence of control signals that brings the system to the equilibrium position (inverted position). This is because, even though the system is time-invariant (i.e., the differential equations governing its dynamics do not change), it is nonlinear. One could linearize the system at various points in the state space, but doing so would make it effectively time-variant.

In addition to the inherent difficulties in controlling nonlinear and time-variant systems, there are optimization problems for which the cost function itself cannot be easily defined. To solve these types of engineering problems, methodologies have been proposed to estimate the cost function (or its dual, the reward signal) and thereby determine the appropriate control policy. Reinforcement Learning-based control methods are currently the state of the art for solving these problems. Some applications include humanoid robots [Peters, Vijayakumar e Schaal 2003], robots with mimicking behavior [Peng *et al.* 2018], and control through visual feedback using RL [Wang *et al.* 2025].

2.2 Elements of Reinforcement Learning

Reinforcement Learning (RL) uses terminology that is slightly different from that of other control paradigms. The two main elements of an RL-based control system are the agent and the environment, which correspond respectively to the controller and the plant. In addition to these two main elements, we can identify the control policy, the reward signal, the value function, and, optionally, the environment model [Sutton e Barto 2018]. The interaction between the agent and the environment is illustrated in Figure 1: at each time-step t, the agent takes an action a_t and the environment makes a state transition that the agent sees as the new state s_t and the reward r_t at that time-step.

Figure 1 – Agent-Environment Interaction



Source: Own elaboration (2025)

The control policy $\pi(s)$ is a function of the system's current state s that determines the action (i.e., control signal) to be taken. It is the core of the RL agent, as it dictates its behavior entirely. Generally, the policy can be stochastic, in the sense that it determines the probabilities of taking a specific action; or it can be deterministic, in which case $\pi(s)$ is the action to be taken.

The reward signal defines the objective in a reinforcement learning problem. At each step, the environment returns a numerical value, known as the reward, to the agent. The agent's primary goal is to maximize the accumulated reward over time and in this way, it determines what the agent considers to be better or worse. In biological terms, we can associate rewards with experiences of pleasure or pain, which are the immediate and determining manifestations of the agent's situation in the environment. The reward signal is fundamental for adjusting the agent's policy; if a chosen action yields low reward, the policy can be updated to select a different action in future situations. In general, reward signals can vary stochastically, depending on the environment's state and the actions performed.

While the reward signal determines whether an action is good or bad in the immediate moment, a given action may yield a higher immediate reward but lead the system to states with lower reward, resulting in a lower overall return. The balance of these two possibilities is what is referred to in the literature as the problem of *exploration versus exploitation*, i.e., how much the agent should exploit its current knowledge of the environment to obtain good short-term rewards and how much it should choose a different action to possibly learn of better paths for maximizing its returns.

After a certain sequence of actions, the agent-environment interaction breaks down, and the system is reset. These sequences are called **episodes** and can be thought of as, e.g., the play of a game or the successful control of a plant for a fixed period of time. After each episode, the agent is updated, i.e., trained, to better utilize the experience it has gained. Tasks that have a natural terminal state are called *episodic*, like, for example, a chess match, in which the terminal state is a win, a loss, or a tie; whereas tasks that do not have a clear final state are called continuing tasks. Control tasks are generally of the latter sort, and what is usually done to address this is to truncate each episode if it exceeds a certain number of time-steps.

The final element of reinforcement learning that we should discuss is value functions. Pretty much every Reinforcement Learning algorithm is based on estimating them, and the mathematics that govern these algorithms are deeply connected to Optimal Control. One of them is the state-value function $V(s)$, given by eq. 2.4, which for any given state s returns the expected reward the agent will receive if it follows the optimal policy π. It is a measure of how good it is to be in a particular state, considering the current reward and expected future rewards, discounted by a positive discount factor $\gamma \leq 1$. The smaller the value of γ, the more short-sighted, so to speak, the agent will be.

As a side note, the value function defined in equation 2.4 is called the *state-value function*, but we can talk about a state-value function over a specific policy π, usually denoted with $V_\pi(s)$. However, when we say "value function" without a qualifier, we mean the state-value function under the optimal policy.

$$V(s) \doteq \mathbb{E}_{\pi^*}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s\right]. \tag{2.4}$$

More importantly, one can easily get a relation between the value function at state s and the value function at the next state s' by simply putting the current reward in front of the rest of the sum. This yields the famous Bellman Equation 2.5 for the state-value function, which can be solved using dynamic programming, much like in Optimal Control. In this context, however, there are two main DP algorithms: value iteration and policy iteration, both of which are guaranteed to converge and yield the optimal policy and the state-value function.

$$V(s) = \max_{\pi} \mathbb{E}(R_t + \gamma V(s')). \tag{2.5}$$

The other essential value function used exhaustively in RL is the action-value function $Q(s,a)$. Again, the notation $Q_\pi(s,a)$ refers to the action-value function under a specific policy π, whereas the $Q(s,a)$ without any qualifier usually means the Q-function under the optimal policy.

Instead of being only a function of the current state, it takes in a specific action a, and its output is the expected return (discounted over time) starting at state s, taking action a, and following policy π thereafter. This has the power to encapsulate both the policy and the state-value function in a single mathematical object, as both can be computed by finding the $\arg\max$ and max of $Q_\pi(s,a)$ over the action space, as shown in equations 2.6a and 2.6b.

$$V(s) = \max_{a} Q(s,a), \tag{2.6a}$$

$$\pi(s) = \arg\max_{a} Q(s,a). \tag{2.6b}$$

2.3 Markov Decision Processes

So far, the elements of RL have been explained here without much formalism, but the mathematical model used to describe the environment in RL is called a Markov Decision Process. It is a universal way to formalize any problem tackled with RL methods, regardless of the application; therefore, it merits a brief section. A more complete definition of MDPs can be found in any classic Reinforcement Learning textbook, such as [Sutton e Barto 2018].

The formal definition of a Markov Decision Process $\mathcal{M}$ is the following 5-tuple:

$$\mathcal{M} =< \mathcal{S}, \mathcal{A}, P, R, \gamma >,$$

where $\mathcal{S}$ is the state space, i.e. the set of all states the environment can be in; $\mathcal{A}$ is the action space, the set of all actions the agent can take; $P : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0,1]$ is the state transition probability function, that is, $P(s,a,s')$ is equal to the probability of going from state s to state s' when action a is taken; $R : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function; and γ is the discount factor, which was explained in the previous section.

The name Markov in Markov Decision Process reflects the stochastic nature of MDPs; however, the framework can still be used for describing systems that have deterministic dynamics, as is the case in Control Engineering. The only difference is that in such cases, the state transition probability function is degenerate; i.e., it is equal to a Dirac delta function $P(\cdot|s,a) = \delta_{f(s,a)}(\cdot)$. However, the relevant theorems regarding MDPs still hold for this type of MDP.

Finally, each time the agent interacts with the environment, it gets what is called the transition tuple $(s_t, a_t, r_{t+1}, s_{t+1}, d_{t+1})$, which contains the previous state, the action taken, the next state, and the done flag, which is used to indicate whether the terminal state was reached or not. In applying RL algorithms to RL problems, this tuple is used to store trajectory buffers in each episode for later training. The environment objects in the standard Python package for RL environments, gymnasium [Brockman *et al.* 2016, Towers *et al.* 2024], conveniently return the transition tuples for precisely that purpose.

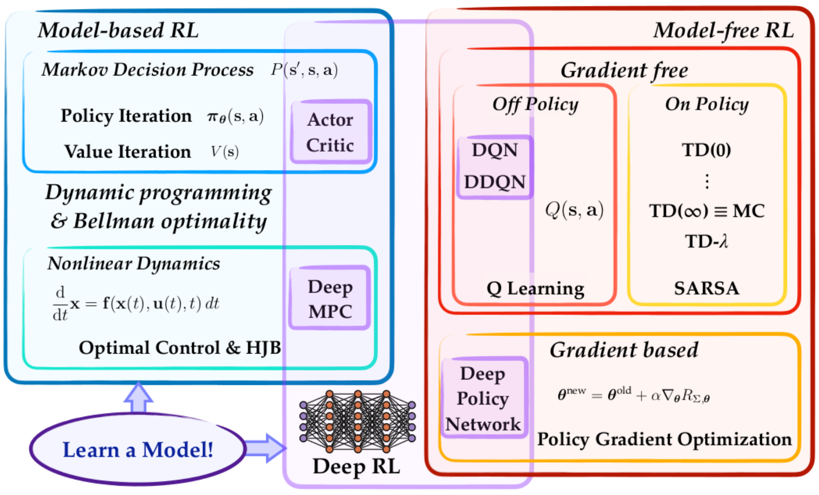
2.4 Overview of RL Techniques

Reinforcement Learning Methods are roughly divided into two categories: model-based RL and model-free RL [Brunton e Kutz 2022]. As the name suggests, model-based methods exploit information about the system's dynamics to find the optimal policy. The aforementioned policy and value-iteration methods, as well as some actor-critic algorithms, are included in this category and will be further discussed in the following sections. Model-free techniques, on the other hand, allow the policy or value function to converge to an optimized solution without informing the agent anything about the system itself. Among these techniques, Q-learning will be used to tackle a handful of control problems.

Model-free methods can be further divided into gradient-free and gradient-based categories, the latter of which includes Policy Gradient Optimization methods and Deep Policy Networks. These methods use an estimate of the reward given the policy π_θ parameterized by θ and apply gradient descent to improve the agent [Brunton e Kutz 2022]. Gradient-free techniques include Q-learning and Temporal Difference Learning; the former of which does not evaluate the policy directly and is thus classified as off-policy, whereas TD learning methods are on-policy. The distinction between the two lies in whether the agent follows the policy learned during training episodes or uses a different, suboptimal policy to better explore the state space.

Finally, neural networks have been shown to be powerful tools for implementing approximate solution methods, and virtually every tool in each category has a version that uses deep neural networks, such as Deep Q-Learning, Deep Policy Networks, Deep Model Predictive Control, and Actor-Critic algorithms. A summary of this rough categorization is illustrated in Figure 2.

Figure 2 – Categorization of Reinforcement Learning techniques


Model-based RL
Markov Decision Process $P(s', s, a)$
Policy Iteration $\pi_\theta(s, a)$
Value Iteration $V(s)$
Actor Critic
Dynamic programming & Bellman optimality
Nonlinear Dynamics
$\frac{d}{dt}x = f(x(t), u(t), t)\, dt$
Deep MPC
Optimal Control & HJB
Learn a Model!
Deep RL
Model-free RL
Gradient free
Off Policy
DQN
DDQN
$Q(s, a)$
Q Learning
On Policy
TD(0)
TD(∞) ≡ MC
TD-λ
SARSA
Gradient based
Deep Policy Network
$\theta^{new} = \theta^{old} + \alpha\nabla_\theta R_{\Sigma,\theta}$
Policy Gradient Optimization

Source: Data Driven Science & Engineering [Brunton e Kutz 2022]

This overview of the techniques is far from complete, as Reinforcement Learning is a rapidly growing field. Greater emphasis was placed on the set of methods to be employed in this work, and the coming sections of this chapter are dedicated to explaining them.

2.5 Dynamic Programming

Dynamic Programming is an iterative process in which the Bellman equation is solved starting from the terminal state and moving backwards to compute an estimate of either the optimal value function or the optimal policy. The process iterates until the value function or policy converges to the true optimal solution. It is strange that a backward-oriented iterative process would have anything to do with Reinforcement Learning techniques: methods that involve going forward along the state space and learning the optimal policy from experience, but DP is indeed the foundation and inspiration of other RL algorithms that have come about [Kaelbling, Littman e Moore 1996].

The two primary DP methods for reinforcement learning are value iteration and policy iteration, and both rely on the fact that for any environment modeled by a Markov Decision Process, there exists an optimal deterministic policy [Bellman 1957].

2.5.1 Value Iteration

Value iteration is a simple iterative process, described in pseudocode 2.1. First, the value function $V(s)$ and an auxiliary value function $V'(s)$ are initialized appropriately. Since its true value will be the discounted sum of the reward when following the optimal policy $\pi'(s)$, it is convenient to start it as the lowest possible signed float value available, and to start $V'(s)$ as anything that allows the loop to start. Inside the loop, the action-value function is computed for the states that come before the terminal state, and for every available action, the value function is updated at that state by maximizing $Q(s, a)$ over the action space, and this is repeated for the previous state again and again until all of them have been evaluated.

This update loop is itself repeated until the difference between two successive value functions is close enough, and it has been shown [Williams e Baird 1993] that when this difference is less than a given ϵ, the computed value function differs from the value function of the optimal policy by $2\epsilon\gamma/(1-\gamma)$.

```
initialize $V(s)$ and $V'(s)$
loop until $|V(s) - V'(s)| < \epsilon$
    V'(s) := V(s)
    loop for $s \in S$
        loop for $a \in A$
            $Q(s,a) := R(s,a) + \gamma \sum_{s' \in S} P(s,a,s')V(s')$
        end loop
        $V(s) := \max_a Q(s,a)$
    end loop
end loop
```

Algorithm 2.1 – Value Iteration Pseudocode

In the equation for the action-value function, $P(s, a, s')$ is a transition probability of going from state s to state s' when action a is taken. If the system under consideration is deterministic, the argument s' in $V(s')$ would just need to be substituted by the state to which the system is brought when it is in the state s and the agent takes action a.

As a final note, the complexity of each loop over all states is $\mathcal{O}(|\mathcal{S}||\mathcal{A}|)$, where $|\mathcal{S}|$ and $|\mathcal{A}|$ are the sizes of the state space and action space, respectively. Still, the overall time complexity of the algorithm depends on how quickly the value function converges.

2.5.2 Policy Iteration

Instead of computing the value function to then find the optimal policy, Policy Iteration manipulates the policy function directly. The pseudocode for policy iteration is given by pseudocode 2.2.

First, a policy π' is initialized randomly, then, in the loop, a copy of the policy is made, and its value function is computed by solving the Bellman equation of the value function. Then, at each state, the policy is updated to the action with the highest expected value. At each step, the performance of π' strictly improves, and when the policy no longer changes, it is guaranteed to be optimal [Kaelbling, Littman e Moore 1996].

```
choose a random policy $\pi'$
loop
    $\pi := \pi'$
    compute value function of policy $\pi$
        solve the linear equations:
            $V_\pi(s) = R(s,\pi(s)) + \gamma \sum_{s' \in S} P(s,\pi(s),s') V_\pi(s')$
    update the policy at each state:
        $\pi'(s) := \arg\max_a (R(s,a) + \gamma \sum_{s' \in S} P(s,\pi(s),s') V_\pi(s')$
end loop when $\pi = \pi'$
```

Algorithm 2.2 – Policy Iteration Pseudocode

2.6 Q-Learning

As previously mentioned in section 2.2, the action-value function $Q_\pi(s, a)$, also known as the action-value function, represents the value of being in state s, taking action a, and following policy π thereafter. Much like the value function, the Q-function can also be written in recursive form, as shown in equation 2.7.

$$Q(s,a) = R(s,a) + \sum_{s' \in \mathcal{S}} P(s,a,s') \max_{a'} Q(s',a'). \tag{2.7}$$

If the system is deterministic, the sum over all possible next states can be dropped, and the equation simplifies to:

$$Q(s,a) = R(s,a) + \max_{a'} Q(s',a').$$

2.6.1 Tabular Q-Learning

Watkins' Q-learning algorithm, proposed in a paper of the same name [Watkins e Dayan 1992], uses a learning rule very similar to temporal difference learning. The learning rule is given by equation 2.8, where r is the reward received in the current time, γ is the discount factor, $\alpha < 1$ is the learning rate, and the term multiplied by α is called the temporal difference error.

$$Q_\pi(s,a) := Q_\pi(s,a) + \alpha(r + \gamma \max_{a'} Q_\pi(s',a') - Q_\pi(s,a)). \tag{2.8}$$

If the learning rate were to be set to 1, $Q_\pi(s, a)$ and $-\alpha Q_\pi(s, a)$ would cancel out, and the agent would consider the current reward along with the quality of the next state. With lower learning rates, it updates the current value more slowly, which is essential for ensuring convergence and stability. Q_π is guaranteed to converge to Q, the action-value function of the optimal policy, if the learning rule is run an infinite number of times over an infinite run, as long as α decays appropriately [Watkins e Dayan 1992].

This simplest form of Q-learning and other methods are called *tabular* in the sense that one can make a table over all states and all possible actions and fill in the value of $Q(s, a)$ for each tuple. For a large state space, or worse, a continuous state space, this would require significant memory. When implementing Q-learning for control in a context where the state space is continuous, other function approximation methods must be employed. For example, defining the action value function as the inner product between a vector of trainable weights w and a feature map of the state variables and action signal. For the swing-up task of a pendulum, where the action a was the torque applied to the shaft, θ is the angle of the pendulum relative to the y-axis, and $\dot{\theta}$ is the angular velocity, it was found through trial and error that the feature map $x = \begin{bmatrix}1 & 1 & \theta^2 & \theta & \dot{\theta}^2 & \dot{\theta} & (a\theta)^2 & a\theta & (a\dot{\theta})^2 & a\dot{\theta}\end{bmatrix}$ was sufficient to enable the agent to find a satisfactory policy for this non-linear control problem [Ghio e Ramos 2019].

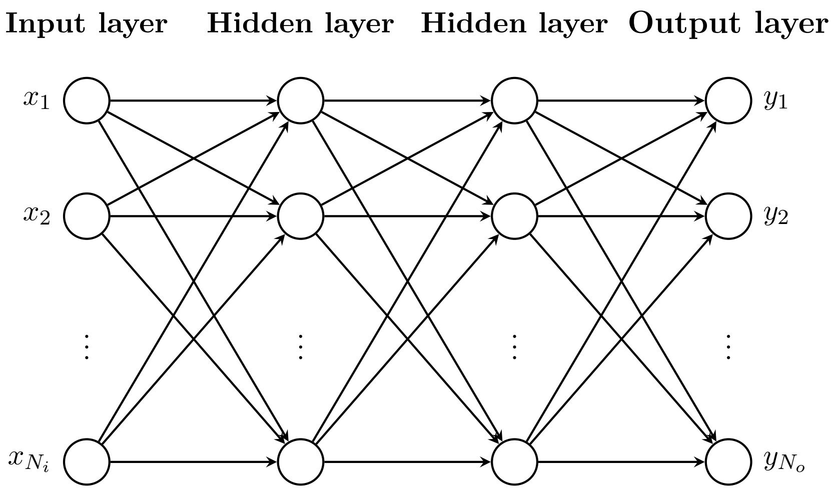
Furthermore, Q-learning is said to be exploration-intensive, in that as long as all action-state pairs are sufficiently explored, it will converge to an optimized Q-function. In other words, it does not matter how the agent behaves, as long as a representative amount of training data is collected throughout the episodes [Kaelbling, Littman e Moore 1996].

One way to achieve this is to use an ϵ-greedy approach. At any given state, a random action is taken with probability ϵ and the best action according to the current values of the Q-function is taken with the remaining probability $1 - \epsilon$ [Sutton e Barto 2018]. ϵ can be set to a high probability in the first few episodes and decay over time, ensuring exploration and later exploiting the agent's knowledge to access states that would not be accessible otherwise. This approach is not unique to Q-learning; it is just an ad hoc technique that can be used with several different RL methods.

2.6.2 Deep Q-Learning

As stated in section 2.4, neural networks have been proven to be excellent tools for approximating value functions and policies in Reinforcement Learning. That is because multilayer perceptrons, i.e., neural networks, are universal function approximators [Hornik, Stinchcombe e White 1989], serving as the Q-function or policy function of nonlinear systems or systems with highly nonlinear reward signals. Figure 5 shows the diagram of a multilayer perceptron.

Figure 3 – Diagram of a Neural Network



Source: Own elaboration (2025)

To use neural networks for value-based Reinforcement Learning, one must choose an architecture (i.e., the number of convolutional and fully connected layers), define a loss function, set learning hyper-parameters such as the learning rate and optimizer (all options of which are, of course, based on backpropagation), and specify the network's training environment. In the case of Deep Q-Learning, it is essential to set a replay memory buffer for a number of episodes from which samples are taken for training, be it randomly or uniformly; this allows the DQN to be trained in an off-policy manner, improving data efficiency [Wang *et al.* 2022].

In "Human-level control through deep reinforcement learning" [Mnih *et al.* 2015], the authors introduced a setup to reduce training instability: having two neural networks with the same number of trainable parameters; the main network Q and the target network Q'. The main network is updated in every training step and is used to determine the policy of the agent in each episode, whereas the target network is kept unchanged, but every certain number of training steps, it is updated to match the main network. The target network gives stability to the training process by introducing a delay in the feedback that inherently exists in Q-Learning, since the expected return needs a Q-function to be evaluated and since this expected return goes into the loss function, as given in equation 2.9. The Deep Q-Learning method described here can be summarized in pseudocode 2.3.

$$y = R + \gamma \max_a Q'(s_{t+1}, a). \tag{2.9}$$

```
Initialize experience replay buffer $D$;
Initialize $Q$ network with random weights $\theta$ and
target network $Q'$ with weights $\theta' = \theta$;
for episode=1, 2...N do
    Reset environment and initial state $\phi(s_0)$;
    for t=0, 1, 2...T do
        Use $\phi(s_t)$ as input of $Q$ network and get $Q$ value of each
           action;
        Choose action $a_t = \epsilon - greedy(s_t)$;
        Take action $a_t$ under $s_t$, observe reward $R$ and new state
           $\phi(s_{t+1})$;
        Push $(\phi(s_t), a_t, R, \phi(s_{t+1}))$ into $D$;
        Randomly take m samples $(\phi(s_j), a_j, R_j, \phi(s_{j+1}))$ from $D$;
        Compute approximate value:
            $y_j = \begin{cases} R_j & \phi(s_{j+1}) \text{ is terminal} \\ y_{nt} & \phi(s_{j+1}) \text{ is nonterminal} \end{cases}$
        where
            $y_{nt} = R_j + \gamma \max_a Q'(\phi(s_{j+1}), a; \theta')$
        Update $Q$ network by executing gradient descent
        on loss function:
            $L_\theta = \frac{1}{m}\sum_{j=1}^{m}(y_j - Q(\phi(s_j), a_j; \theta))^2$
        Set $\theta' = \theta$ every C steps;
    end
end
```

Algorithm 2.3 – Deep Q-Learning Algorithm

A more sophisticated way to update the target network is to use a soft update, first proposed in a work on Deep Deterministic Policy Gradient [Lillicrap *et al.* 2015], an actor-critic algorithm discussed in the next sections. Instead of copying the main network every C number of steps, the target network parameters θ' are always slowly updated according to equation 2.10, where $0 < \tau < 1$ is the update rate.

$$\theta' := \tau\theta + (1-\tau)\theta'. \tag{2.10}$$

Usually, the DQN's input layer takes a state-characterizing input, whereas the output layer has as many outputs as the action space. One could make it so that the action signal goes into the input layer, but then the network would have a single neuron on the output, and there would still be the problem with having to compute a set number of actions for each state in order to find $\arg\max_a Q(s,a)$. For this reason, Deep Q-Learning has significant problems with continuous action spaces, and that is something that can be avoided by choosing an actor-critic algorithm instead.

2.7 Actor-Critic Algorithms

Actor-Critic refers to a set of different algorithms that have one thing in common, namely, that inside the agent there exists a function that chooses the action to be taken and another function that evaluates how good this action is. They are a combination of value-based RL and policy-based RL and at first glance this appears to be an unnecessary added complexity; however, the Critic in Actor-Critic Algorithms tends to improve the variance and speed the learning process [Sutton e Barto 2018] while having a function approximator for the policy itself enables finding a policy over a continuous action space, as will be discussed in the DPG section.

2.7.1 Deterministic Policy Gradient

The Deterministic Policy Gradient Algorithm is a type of actor-critic algorithm that approximates a policy over a continuous action space without discretization, unlike Q-learning or DQNs, which require it to handle non-discrete action spaces. For a problem in which the action space has 7 degrees of freedom, like in a robotic arm for example, a very course discretization, say, three possible values for each control variable, the discrete action space would have a cardinality of $3^7 = 2187$ [Lillicrap *et al.* 2015], which would require a considerable amount of exploration for a value function to be learned. DPG and DDPG (Deep Deterministic Policy Gradient) algorithms solve that problem by using a separate function approximator for the policy, which is updated using the gradient of the policy with respect to its parameters, which is proportional to the $Q(s, a)$ function. This result is called the Policy Gradient Theorem [Sutton *et al.* 1999], and it is more clearly stated in the following subsection.

2.7.1.1 Policy Gradient Theorem

Given a policy $\pi(s, a, \theta)$, where θ is a parameter vector that characterizes the policy, that acts on a Markov Decision Process with state transition probability $P(s, a, s')$ and expected reward $R(s, a)$, the objective of the MDP can be written as in equation 2.11, where $d_\pi(s)$ is the discounted stationary state distribution under policy π, i.e., it is the fraction of time the environment is at state s when the agent follows policy π discounted by the discount factor γ: $d_\pi(s) = (1 - \gamma)\sum_{t=0}^{\infty} \gamma^t \Pr(s_t = s \mid \pi)$.

$$V(s) = E\left\{\sum_{t=1}^{\infty} \gamma^{t-1} r_t | s_0, \pi\right\} = \sum_s d_\pi(s) \sum_a \pi(s, a) R(s, a). \tag{2.11}$$

Given this objective, the state-action value function is.

$$Q_\pi(s,a) = E\left\{\sum_{k=1}^{\infty} \gamma^{k-1} r_{t+k} | s_t = s, a_t = a, \pi\right\}. \tag{2.12}$$

The Policy Gradient Theorem states that the objective's derivative with respect to the policy's parameter is as given in equation 2.13.

$$\frac{\partial V}{\partial \theta} = \sum_s d_\pi(s) \sum_a \frac{\partial \pi(s,a)}{\partial \theta} Q_\pi(s,a). \tag{2.13}$$

The proof for this theorem can be found in [Sutton e Barto 2018] chapter 13.2 and in the original paper [Sutton *et al.* 1999]. The relevance of the theorem lies in the fact that the gradient of the objective with respect to the policy parameters is directly proportional to the average of the Q-function. Therefore, it can be used as a loss to improve the policy progressively. In that sense, the Q-function is a critic of the policy function, or an actor.

2.7.1.2 DDPG algorithm

The Deep Deterministic Policy Gradient algorithm, as the name suggests, is an implementation of DPG using Deep Neural Networks, largely inspired by the success achieved with DQNs [Lillicrap *et al.* 2015]. The architecture of the value function neural network has a total number of input neurons equal to the dimension of the observation space plus the degrees of freedom of the policy (i.e., the dimension of the action space). In contrast, it has only a single output neuron, since the value function is a scalar. The policy network, on the other hand, has as many input neurons as there are dimensions in the observation space and as many output neurons as there are dimensions in the action space.

As in Deep Q-Learning Algorithms, there are target networks in addition to the main networks used for action selection. As stated previously, the target networks are used to compute the loss functions for the actor and critic networks, and their parameters are updated via soft updates, which stabilize learning. The soft update given by equation 2.10 is essentially a weighted average and can be thought of as a low-pass filtering of the main network's parameters. Since the learning rule includes feedback, it becomes clear why this filtering improves training stability.

Finally, the learning rule of the Critic Network is a gradient descent in the Bellman Loss (i.e., the Mean Squared Error of the value of Q given by the main network with respect to the discounted return computed by the target network), and the learning rule of the Actor Network is a gradient ascent of the objective, which has been shown to be proportional to the action-value function. pseudocode 2.4 summarizes the entire algorithm.

```
Initialize the Critic Network $Q(s,a|\theta^Q)$ and actor Network $\mu(s|\theta^\mu)$
   with parameters $\theta^q$ and $\theta^\mu$
Initialize the target networks with the same parameters
Initialize the Replay Memory R
for episode = 1, ... M do
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial observation state s1
    for t = 1, ... T do
        Select action at $= \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current
           policy and exploration noise
        Execute action at and observe reward $r_t$ and observe new
           state $s_{t+1}$
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in R
        Sample a random minibatch of N transitions $(s_i, a_i, r_i, s_{i+1})$
           from R
        Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
        Update critic by minimizing the Bellman loss:
           $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
        Update the actor policy using the sampled policy
           gradient:
            $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i Q(s, \mu(s|\theta^\mu)|\theta^Q)$

        Perform soft update on the target networks
        $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
        $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$

    end
end
```

Algorithm 2.4 – Deep Deterministic Policy Gradient Algorithm

2.7.2 Twin Delayed Deep Deterministic Policy Gradient

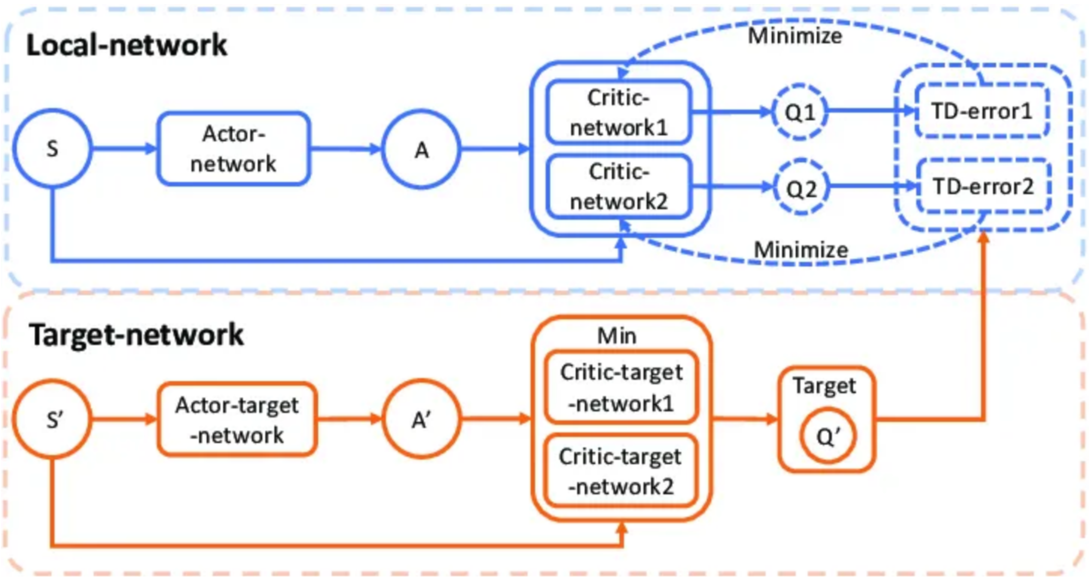
Twin Delayed Deep Deterministic Policy Gradient [Fujimoto, Hoof e Meger 2018] is a variant of DDPG with an architecture very similar to the original scheme. There is a target actor-critic that is updated more slowly according to the update rule in equation 2.10, which is used to compute the actor gradient loss and the critic Bellman loss. Additionally, there is a main actor critic, which is actually responsible for choosing the policy throughout the training process. DDPG alone works well for many systems; however, it still suffers from stability problems during training, which were observed in some test cases and are included in the results (section 3.2).

TD3 was developed to address precisely these instability problems, and it differs from DDPG in three main aspects:

- Clipped Double-Q Learning. The TD3 agent learns two Q-functions and uses the least of the values computed by them in the target critic network to update the main one.
- Delayed Policy Update. The policy is only updated once every two or three value function updates.
- Target Policy Smoothing: TD3 introduces noise to the target policy's action to smooth the Q-function. This regularization makes it harder for the policy to exploit errors in the value estimation.

These differences are nicely summarized in figure 4, and the detailed description of the algorithm is in pseudocode 2.5.

Figure 4 – TD3 framework



Source: A-td3: An adaptive asynchronous twin delayed deep deterministic for continuous action spaces [Wu *et al.* 2022]

The source of instability in DDPG is the accumulation of error in the temporal difference estimate. Although one can expect minor errors in the TD estimate during a single update of the agent, these errors can accumulate over time, leading to large overestimations of the value and suboptimal policies [Fujimoto, Hoof e Meger 2018].

```
Initialize critic networks $Q_1(s,a|\theta^{Q_1})$, $Q_2(s,a|\theta^{Q_2})$ and actor network
   $\mu(s|\theta^{\mu})$ with parameters $\theta^{Q_1}$, $\theta^{Q_2}$, and $\theta^{\mu}$
Initialize target networks $Q'_1$, $Q'_2$, $\mu'$ with parameters
        $\theta^{Q'_1} \leftarrow \theta^{Q_1}$,
        $\theta^{Q'_2} \leftarrow \theta^{Q_2}$,
        $\theta^{\mu'} \leftarrow \theta^{\mu}$.
Initialize replay buffer $\mathcal{R}$
for episode = 1, ..., M do
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial state $s_1$
    for t = 1, ..., T do
        Select action with exploration noise:
        $a_t = \mu(s_t|\theta^{\mu}) + \epsilon$, where $\epsilon \sim \mathcal{N}(0, \sigma)$
        Execute action $a_t$, observe reward $r_t$, and new state $s_{t+1}$
    Store transition $(s_t, a_t, r_t, s_{t+1})$ in $\mathcal{R}$
        if ready to update then
            Sample minibatch of N transitions $(s_i, a_i, r_i, s_{i+1})$ from $\mathcal{R}$
            # Target policy smoothing
            $\tilde{a}_{i+1} = \texttt{clip}(\mu'(s_{i+1}|\theta^{\mu'}) + \texttt{clip}(\epsilon', -c, c), a_{low}, a_{high})$,
            where $\epsilon' \sim \mathcal{N}(0, \sigma')$
            # Compute target Q-value using the smaller of the
               two target critics
            $y_i = r_i + \gamma \min_{j=1,2} Q'_j(s_{i+1}, \tilde{a}_{i+1}|\theta^{Q'_j})$
            # Update both critics by minimizing the MSE loss
            $L(\theta^{Q_j}) = \frac{1}{N}\sum_i (Q_j(s_i, a_i|\theta^{Q_j}) - y_i)^2$, for j = 1, 2
            Update $\theta^{Q_1}$ and $\theta^{Q_2}$ using gradient descent
            # Delayed policy updates
            if t mod d == 0 then
                Update actor policy by gradient ascent:
                    $\nabla_{\theta^{\mu}} J \approx \frac{1}{N}\sum_i \nabla_a Q_1(s, a|\theta^{Q_1})|_{a=\mu(s)} \nabla_{\theta^{\mu}} \mu(s|\theta^{\mu})$
                # Soft update target networks
                $\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}$, $\theta^{Q'_1} \leftarrow \tau\theta^{Q_1} + (1-\tau)\theta^{Q'_1}$,
                    $\theta^{Q'_2} \leftarrow \tau\theta^{Q_2} + (1-\tau)\theta^{Q'_2}$


    end
end
```

Algorithm 2.5 – Twin Delayed Deep Deterministic Policy Gradient (TD3) Algorithm

One way to compare the stability of DDPG and TD3 is to compute the mean Q-values of each network during training. A clear sign of instability and Q-value overestimation is when the Q-values suddenly jump from one training episode to another. One instance of this problem was observed during training a DDPG agent to stabilize the cartpole system: the Q-value jumped from 81.6 to 306.3 (well above the maximum possible return in an episode for that environment), and then it started oscillating erratically. The Q-values of the TD3 agent, on the other hand, were stable.

2.8 Choosing a Reward Signal

One of the most challenging things in reinforcement learning is designing a reward signal that encourages the agent to learn to do what you want it to do. In episodic tasks, in which the terminal state is, so to speak, the objective state, one could give a reward of 1 for reaching the terminal state and a reward of 0 in every other state. Because of the discount factor, the agent will eventually learn which actions lead to achieving the terminal state after a certain number of successful episodes, allowing it to determine whether it is going in the right direction throughout each episode. The problem is that it has to reach the terminal state to get any feedback on whether its actions were good. This is an extreme example of what the literature calls a sparse reward.

Sparse rewards are suboptimal because they generally do not provide enough feedback for the agent to learn quickly what it needs to do. Luckily, in the context of RL applied to control engineering, it is easiest to define the reward as minus the step cost function $L(s, u)$ (the same one that constitutes the cost that is minimized in optimal control: $J = \sum\limits_{t} L(s_t, u_t)$), and this is a non-sparse reward signal since it is a continuous function of the state variables and the control signal (see equation 2.1).

Still, one might want to give the agent a reward bonus when it does something right to incentivize certain actions and converge to the optimal policy more quickly; however, this needs to be done with caution. Sometimes, when certain bonuses are added, the agent can *reward hack* its way into maximizing the total shaped reward without actually solving the original, intended MDP. While training a robot to ride a bicycle, an RL agent was given a bonus whenever it moved closer to the goal [Randløv e Alstrøm 1998]. The robot then learned to ride in circles since no punishment was given when it moved away from the goal. Another such case involved a soccer-playing robot that was given a positive reward for touching the ball. What it then learned to do was to trap the ball in a corner and vibrate near it to get as many bonuses as possible [Ng, Harada e Russell 1999].

Similar reward-hacking bugs were also encountered throughout the production of this work. When training a DDPG to learn the swing-up and balance task for an inverted pendulum, the reward was $-L$, i.e., the cost from optimal control theory but with the sign flipped; because the reward was always negative and an episode would end whenever the cart went out of bounds, the agent learned a policy whereby it would swing the pendulum up, start balancing it, and then go as fast as possible to the closest wall to end the episode. It seemed that when an agent only receives negative rewards while having the possibility of ending the run, it would try to kill itself. Indeed, when a constant offset was added to the reward to ensure it was positive most of the time, the agent did not learn this behavior, and we had successfully implemented suicide prevention (for RL agents).

The same thing happened during the training process of the rotary pendulum. At first, there wasn't a condition to stop the run; however, with the reward set to $-L$ and the horizontal arm's state variable being wrapped between $-\pi$ and π, the agent found a sub-optimal local maxima whose policy consisted of rotating very fast in one direction to keep the vertical angle close to $\frac{\pi}{2}$ and its derivative at 0. A stop condition was added, but then the self-destruction reward hacking issue occurred again. However, once more, it was only a matter of adding a positive offset to the reward.

Given the potential negative implications of changing a reward signal that we know works, is there a way to give bonuses to the agent without it leading to reward hacking? It turns out there is, and it was laid out in the 1999 paper *Policy Invariance under reward transformations* [Ng, Harada e Russell 1999]. The trivial policy-invariant transformations are scaling the reward signal and adding a constant to it. If a function has a global maximum, scaling or adding a constant to it will only change the value of the function at that maximum point, but it will not change the point in the domain at which the maximum occurs. The authors showed that any reward bonus in the form given in equation 2.14 will also keep the optimal policy unchanged relative to the original MDP's optimal policy. $F(s, a, s')$ is the reward bonus, γ is the same discount factor of the original MDP, $\phi(x)$ is a *potential* function, and s, a, and s' are the current state, the action taken, and the next state, respectively.

$$F(s, a, s') = \gamma\phi(s') - \phi(s). \tag{2.14}$$

The proof of this property is actually quite simple. The agent is attempting to learn a policy whereby each action taken maximizes the discounted expected return. The discounted return going from state s_1 of the original MDP is $J(s_1) = \sum_{t=1}^{\infty} \gamma^{t-1} r_t$. In the transformed MDP, the reward will have a bonus $F(s_t, a, s_{t+1})$, and so the return will be $J^*(s_1) = \sum_{t=1}^{\infty} \gamma^{t-1}(r_t + \gamma\phi(s_{t+1}) - \phi(s_t)) = \sum_{t=1}^{\infty} \gamma^{t-1} r_t + \sum_{t=1}^{\infty} \gamma^t \phi(s_{t+1}) - \gamma^{t-1}\phi(s_t)$. When the return is broken down into two series, the second series can be recognized as a telescopic series that converges to $-\phi(s_1)$ (this holds as long as $\gamma^t \phi(s_{t+1}) \to 0$, since we usually choose a bounded potential function ϕ, this will be satisfied). As the transformation only amounts to a constant being added to the return, it is a policy-invariant transformation.

This shaping reward was tested in this work; however, because the standard reward function $-L$ in equation 2.1 is already a continuous function, adding a shaping reward was not particularly beneficial. The most relevant heuristics for setting the cost/reward function were 1) giving higher weights for the angle variables in swing-up tasks, 2) ensuring the reward was mostly positive in environments in which the agent can terminate an episode (Cartpole and Rotary Pendulum), and 3) properly scaling the cost term to ensure learning; i.e., if one scales L by a very low scalar, the reward function will be too flat for the agent to notice a gradient.

2.9 Hybrid Control with Reinforcement Learning

Since established control strategies work well in equilibrium positions where linear approximations are valid, an interesting approach is to use a hybrid control law that combines an LQR with a reinforcement learning agent, with the latter handling the nonlinear tasks while the former addresses stabilization problems. For the plants considered in section 2.10, the most sensitive state variable was the angle θ in the simple pendulum, the cartpole, and also in the rotary pendulum system. By 'sensitive', it is meant that as this variable moves away from the linearization point, the linearized dynamics change more rapidly.

The approach we propose consists in making a weighted sum of the LQR control law and the RL agent's output, in which the weights are a linearity factor $\mathrm{LF}(s)$ and its complement, as stated in equation 2.15.

$$u = u_{\mathrm{LQR}} \cdot \mathrm{LF}(s) + u_{\mathrm{RL}} \cdot (1 - \mathrm{LF}(s)). \tag{2.15}$$

The linearity factor is a function of the state that is close to 1 in the equilibrium point and virtually 0 when it is away from it. To ensure a smooth transition, we used the continuous function given in equation 2.16, which transitions from 1 to 0 at about $0.8\mathrm{rad} = 45^\circ$.

$$\mathrm{LR}(\theta) = e^{-2.6\theta^6}. \tag{2.16}$$

Another critical point is that the RL agents themselves are trained without the aid of this hybrid approach, since any action near the equilibrium point yields the same reward, potentially hindering the training process.

2.10 Systems Under Consideration

As mentioned before, the RL environments considered in this work were classic control problems. Namely, the simple pendulum, the pendulum on a cart (Cart Pole) and the rotary pendulum. In the following subsections, only the system descriptions and dynamics are given, while the basic reward signal is borrowed from optimal control theory. The specific weights of each quadratic term in the cost function are mentioned only in the results section, as they varied depending on the agent's goal.

The derivation of the dynamics of each system was computed using the Euler-Lagrange equation, given in equation 2.17, where q_i is a generalized coordinate, $\mathcal{L} = T - V$ is the Lagrangian of the system, and Q_{q_i} is a generalized force that accounts for external forces along the direction of the coordinate q_i. These can include viscous friction forces and, more importantly, the force from an actuator. Once the dynamics of the systems are computed, it is important to isolate the second derivative of each state variable, which is useful for implementing the simulated environment and for computing the LQR gain for the Hybrid RL problems.

$$\frac{d}{dt}\left(\frac{\partial \mathcal{L}}{\partial \dot{q}}\right) - \frac{\partial \mathcal{L}}{\partial q} = Q_q. \tag{2.17}$$

The complete derivation of the system's dynamics was performed only for the cart-pole, while the Lagrangian was explicitly computed for the pendulum system. For the rotary pendulum, a reference paper was cited for the derivation, and only the resulting Lagrangian expression was presented.

Finally, all simulated environments used to test the RL algorithms were implemented as Gymnasium environments [Towers *et al.* 2024] and used adaptive RK4 for numerical integration with a base time step of $\Delta t = 0.025s$. That means that each time-step the agent sees lasts $0.025s$ in the simulation; however, the adaptive algorithm can take more numerous, smaller time-steps if the time derivative is too high.

Another important note is the fact that the observation that the agent has access to is not directly the state vector of the system, e.g. $\vec{x} = \begin{bmatrix} x & \dot{x} & \theta & \dot{\theta} \end{bmatrix}$, but rather $\vec{x} = \begin{bmatrix} x & \dot{x} & \sin\theta & \cos\theta & \dot{\theta} \end{bmatrix}$. Passing the sine and cosine of angle state variables rather than the angles themselves avoids a discontinuity at $\theta = 2\pi$ while at the same time fully informing the agent of the current state of the system. The same was made for the angle ϕ in the rotary pendulum system.

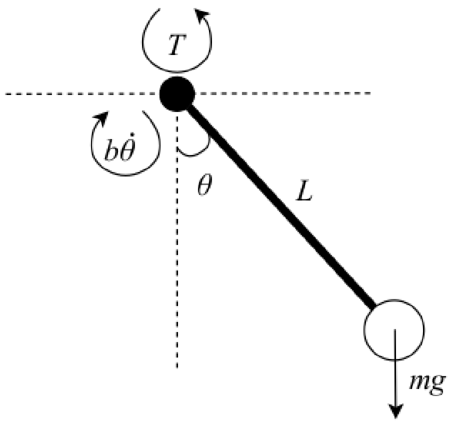
2.10.1 Simple Pendulum

The simple pendulum has a single degree of freedom and thus two state variables: the angle θ the pendulum makes with the vertical axis, such that a clockwise rotation translates to a positive θ and the angular velocity $\dot{\theta}$, as in equation 5.

The agent interacts with the pendulum by sending a one-dimensional action signal u that is equal to the torque applied to the axis of rotation.

The pendulum has a point mass m at its tip and a length l. The height $-l\cos\theta$ of the mass determines the gravitational potential energy $U = -mgl\cos\theta$ of the pendulum, and the kinetic energy is purely rotational and equal to $T = \frac{1}{2}ml^2\dot{\theta}^2$. Thus, the Lagrangian of the system is:

Figure 5 – Diagram of the simple pendulum system



Source: Q-learning-based Model-free Swing Up Control of an Inverted Pendulum [Ghio e Ramos 2019]

$$\mathcal{L} = T - V = \frac{1}{2}ml^2\dot{\theta}^2 + mgl\cos\theta. \tag{2.18}$$

By plugging this expression in equation 2.17 and noting that the external torques are $Q_\theta = \tau - \beta\dot{\theta}$ (where $\tau = u$ is the control torque and β is a viscous friction coefficient), we get the dynamics of the pendulum in equation 2.19.

$$\ddot{\theta} = \frac{\tau - \beta\dot{\theta}}{ml^2} - \frac{g}{l}\sin\theta. \tag{2.19}$$

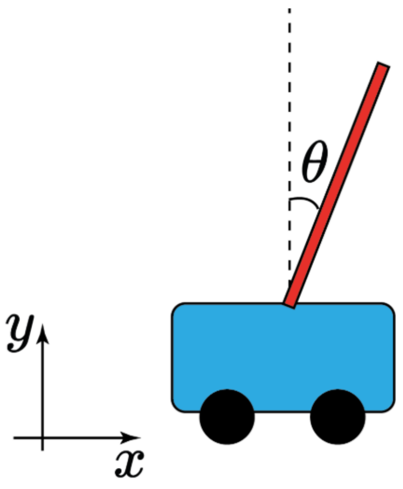
2.10.2 Cart Pole

The state variables of this system are the position x of the cart, the angle θ the pole makes with the vertical axis, and their respective derivatives. For θ, a clockwise rotation has a positive sign, as shown in figure 6. These are sufficient to define the system's potential and kinetic energy at any moment.

If we consider the pole to be a uniformly distributed mass m over a section l and consider the cart to be a point mass M, the kinetic energy can be computed by calculating the energy due to the linear movement of the center of mass of the pole, due to the linear movement of the cart and due to the rotational movement of the pole about its center of mass, as in equation 2.20, where v_c is the speed of the center of mass of the pole and I is the moment of inertia about the center, which for a uniform distribution of mass is $I = \frac{1}{12}ml^2$.

$$\frac{1}{2}M\dot{x}^2 + \frac{1}{2}mv_c^2 + \frac{1}{2}I\dot{\theta}^2. \tag{2.20}$$

Figure 6 – Diagram of the cart pole system



Source: Own elaboration (2025)

Let x_c and y_c be the center of mass of the pole. Since the axis of the pole is located in the origin, $x_c = x + \frac{l}{2}\sin\theta$ and $y_c = \frac{l}{2}\cos\theta$. Taking the derivative of x_c and y_c with respect to time, we get the velocity of the center of mass of the pole:

$$(\dot{x}_c)^2 = (\dot{x} + \frac{l}{2}\dot{\theta}\cos\theta)^2 = \dot{x}^2 + l\dot{x}\dot{\theta}\cos\theta + \frac{l^2}{4}\dot{\theta}^2\cos^2\theta,$$

$$(\dot{y}_c)^2 = (-\frac{l}{2}\dot{\theta}\sin\theta)^2 = \frac{l^2}{4}\dot{\theta}^2\sin^2\theta,$$

$$\therefore v_c^2 = \dot{x}_c^2 + \dot{y}_c^2 = \dot{x}^2 + l\dot{x}\dot{\theta}\cos\theta + \frac{l^2}{4}\dot{\theta}^2.$$

Inserting this result in equation 2.20 and knowing that the potential energy is $V = mg\frac{l}{2}\cos\theta$, we can compute the Lagrangian of the system, given in 2.21.

$$\mathcal{L} = T - V = \frac{1}{2}(m + M)\dot{x}^2 + \frac{ml}{2}\dot{x}\dot{\theta}\cos\theta + \frac{ml^2}{6}\dot{\theta}^2 - mg\frac{l}{2}\cos\theta. \tag{2.21}$$

We can now compute the partial derivatives of this expression to obtain the differential equations governing the system's dynamics.

$$\frac{\partial\mathcal{L}}{\partial\dot{x}} = (M + m)\dot{x} + \frac{ml}{2}\dot{\theta}\cos\theta,$$

$$\frac{d}{dt}\left(\frac{\partial\mathcal{L}}{\partial\dot{x}}\right) = (M + m)\ddot{x} + \frac{ml}{2}(\ddot{\theta}\cos\theta - \dot{\theta}^2\sin\theta),$$

$$\frac{\partial \mathcal{L}}{\partial x} = 0.$$

The generalized force in the x direction is the viscous force $-b\dot{x}$ plus the control signal u.

$$(M+m)\ddot{x} + \frac{ml}{2}\ddot{\theta}\cos\theta = \frac{ml}{2}\dot{\theta}^2\sin\theta + u - b\dot{x}. \tag{2.22}$$

Now we do the same for the generalized coordinate θ:

$$\frac{\partial \mathcal{L}}{\partial \dot{\theta}} = \frac{ml}{2}\dot{x}\cos\theta + \frac{ml^2}{3}\dot{\theta},$$

$$\frac{d}{dt}\left(\frac{\partial \mathcal{L}}{\partial \dot{\theta}}\right) = \frac{ml}{2}(\ddot{x}\cos\theta - \dot{x}\dot{\theta}\sin\theta) + \frac{ml^2}{3}\ddot{\theta},$$

$$\frac{\partial \mathcal{L}}{\partial \theta} = mg\frac{l}{2}\sin\theta - \frac{ml}{2}\dot{x}\dot{\theta}\sin\theta.$$

Since the only external force is the viscous friction torque $-\beta\dot{\theta}$, we can plug these results into the Euler-Lagrange equation to get 2.23

$$\frac{ml}{2}\ddot{x} + \frac{1}{3}ml^2\ddot{\theta} = mg\frac{l}{2}\sin\theta - \beta\dot{\theta}. \tag{2.23}$$

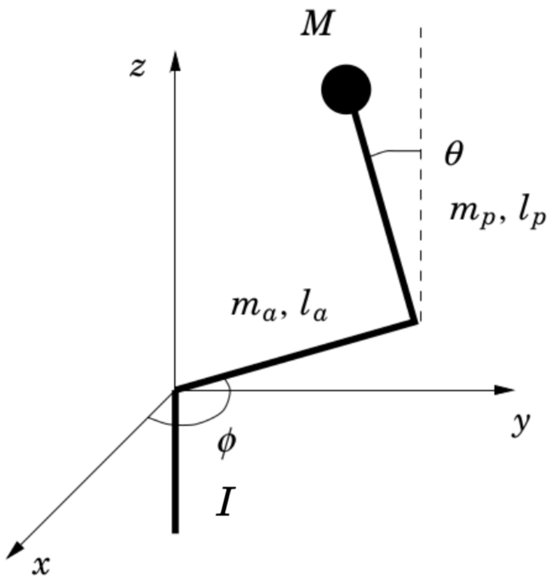
Equations 2.22 and 2.23 can then be combined in compact vector form, given by eq. 2.24.

$$\begin{bmatrix} M+m & \frac{ml}{2}\cos\theta \\ \frac{ml}{2}\cos\theta & \frac{1}{3}ml^2 \end{bmatrix} \begin{bmatrix} \ddot{x} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} F + \frac{ml}{2}\dot{\theta}^2\sin\theta - b\dot{x} \\ mg\frac{l}{2}\sin\theta - \beta\dot{\theta} \end{bmatrix}. \tag{2.24}$$

2.10.3 Rotary Pendulum

There is an excellent paper on deriving the dynamics of the rotary pendulum [Gäfvert 1998]. The pendulum consists of a point mass M held at the tip of the vertical arm, with length l_p and uniformly distributed mass m_p that is attached to the horizontal arm of length l_a and mass m_a. This horizontal arm is driven by a control torque $\tau = u$ that drives a central pillar with a moment of inertia I. The same sign convention is used in this work: the angle ϕ that the horizontal arm makes with the x-axis is positive in the counter-clockwise direction (when viewed from above). The angle θ that the vertical arm makes with the z-axis is positive when it goes in the anti-clockwise direction, as in the diagram of figure 7.

Figure 7 – Diagram of the rotary inverted pendulum system



Source: Modelling the Furuta Pendulum [Gäfvert 1998]

The Lagrangian of the pendulum can be found by dividing the potential and kinetic energies into four parts, each of which is due to the center pillar's moment of inertia I, the point mass M or the distributed masses m_a and m_p. The total potential energy is given in equation 2.25, and the kinetic energy is described in equation 2.26.

$$\begin{aligned} V &= V_c + V_a + V_p + V_m \\ &= 0 + 0 + \tfrac{1}{2} m_p g l_p \cos\theta + M g l_p \cos\theta \\ &= \left(M + \tfrac{1}{2} m_p\right) g l_p \cos\theta. \end{aligned} \tag{2.25}$$

$$\begin{aligned} T &= T_c + T_a + T_p + T_m \\ &= \tfrac{1}{2} I \dot{\phi}^2 + \tfrac{1}{6} m_a l_a^2 \dot{\phi}^2 + \tfrac{1}{2} m_p \left[(l_a^2 + \tfrac{1}{3} l_p^2 \sin^2\theta) \dot{\phi}^2 + l_a l_p \cos\theta \, \dot{\phi}\dot{\theta} + \tfrac{1}{3} l_p^2 \dot{\theta}^2 \right] \\ &\quad + \tfrac{1}{2} M \left[(l_a^2 + l_p^2 \sin^2\theta) \dot{\phi}^2 + 2 l_a l_p \cos\theta \, \dot{\phi}\dot{\theta} + l_p^2 \dot{\theta}^2 \right]. \end{aligned} \tag{2.26}$$

Plugging these terms into the Euler-Lagrange equation for both degrees of freedom and setting $Q_\phi = \tau = u$ and $Q_\theta = 0$ we get the dynamics of the system in matrix form:

$$D(\phi,\theta) \begin{bmatrix} \ddot{\phi} \\ \ddot{\theta} \end{bmatrix} + C(\phi,\theta,\dot{\phi},\dot{\theta}) \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \end{bmatrix} + g(\phi,\theta) = \begin{bmatrix} \tau \\ 0 \end{bmatrix}, \tag{2.27}$$

Where the matrices $D(\phi,\theta)$ and $C(\phi,\theta,\dot{\phi},\dot{\theta})$ and the vector $g(\phi,\theta)$ are defined below:

$$D(\phi, \theta) \triangleq \begin{bmatrix} \alpha + \beta \sin\theta^2 & \gamma \cos\theta \\ \gamma \cos\theta & \beta \end{bmatrix},$$

$$C(\phi, \theta, \dot{\phi}, \dot{\theta}) \triangleq \begin{bmatrix} \beta \cos\theta \sin\theta \dot{\theta} & \beta \cos\theta \sin\theta \dot{\phi} - \gamma \sin\theta \dot{\theta} \\ -\beta \cos\theta \sin\theta \dot{\phi} & 0 \end{bmatrix},$$

$$g(\phi, \theta) = \begin{bmatrix} 0 \\ -\delta \sin\theta \end{bmatrix}.$$

And the shorthand variables α, β, γ and δ are:

$$\begin{aligned} \alpha &= I + \Big(M + \tfrac{m_a}{3} + m_p\Big) l_a^2, \\ \beta &= \Big(M + \tfrac{m_p}{3}\Big) l_p^2, \\ \gamma &= \Big(M + \tfrac{m_p}{2}\Big) l_a l_p, \\ \delta &= \Big(M + \tfrac{m_p}{2}\Big) g l_p. \end{aligned}$$

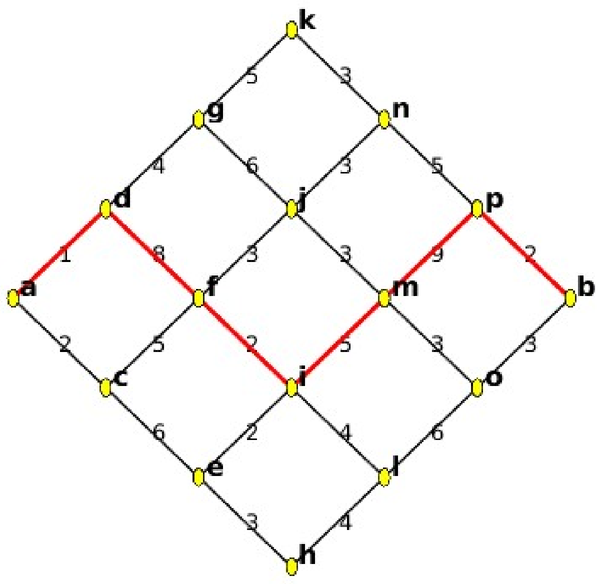
In choosing the system's parameters (or designing a real controllable plant), it is essential that the determinant of the mass matrix $D(\phi, \theta)$ be non-zero at the linearization point, since when it is zero, the two state variables are decoupled. One cannot reliably control θ by applying a torque in ϕ. Indeed, when $\alpha = \beta = \gamma = \delta = 1$ is chosen, the system is unstable [Gäfvert 1998].

3 COMPUTATIONAL EXPERIMENTS

3.1 Solving a simple discrete MDP with DP and Q-Learning

A straightforward way to compare dynamic programming to Q-learning is to use a very small deterministic MDP, which can be represented as a graph or an adjacency matrix. In the graph representation, each node represents a state, and each link represents a possible transition; the number on the link indicates the reward obtained by taking that transition. The graph can be described by an adjacency matrix M, where each entry m_{ij} is the reward for transitioning from state i to state j, and if the entry is 0, then there is no connection between node i and node j. The matrix description was used to parse the MDP for the agent. The graph of the particular MDP used in this test is given in Figure 8, where a is the node of the starting state, b is the node of the terminal state, and each action is a simple binary choice between going up or down in the graph.

Figure 8 – Graph of the MDP with the optimal path traced (in red)



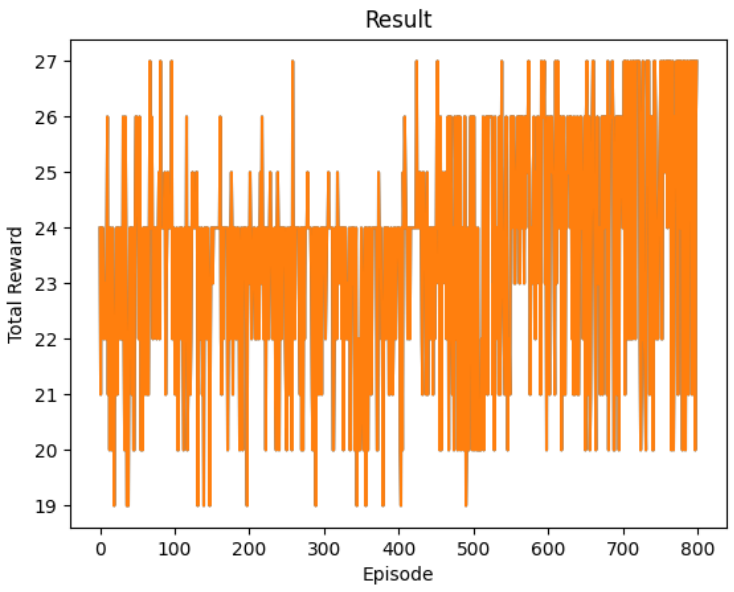
Source: Own elaboration (2025)

One can check that the optimal path yields a reward of 27. Using Value iteration, the value function table was computed, and the optimal path (also in figure 8) was obtained.

A Q-network with two hidden layers (each containing 32 neurons) was trained to learn the optimal policy's value function. Each state was represented as a 16-entry array, with each entry set to 0 except for the one corresponding to the current state, which was set to 1. Thus, the network's input layer has 16 neurons. Finally, the output layer has two neurons: one for moving up and the other for moving down through the graph. The training process was the Deep Q-learning algorithm described in section 2.6.2.

By the end of the training process, the value function can be obtained by maximizing the Q-network's output, and the optimal action at each state can be found by taking the argmax. Figure 9 shows the rewards gained at each episode, and figure 10 shows the mean difference between the Q network's highest output and the value function found through DP (i.e. $\frac{1}{N}\sum_s |\max_a Q_\theta(s) - V(s)|$). This error was less than 1 by the end of the training process, and since all rewards are integer-valued, maximizing the Q-function at each state indeed yields the highest possible return of 27.

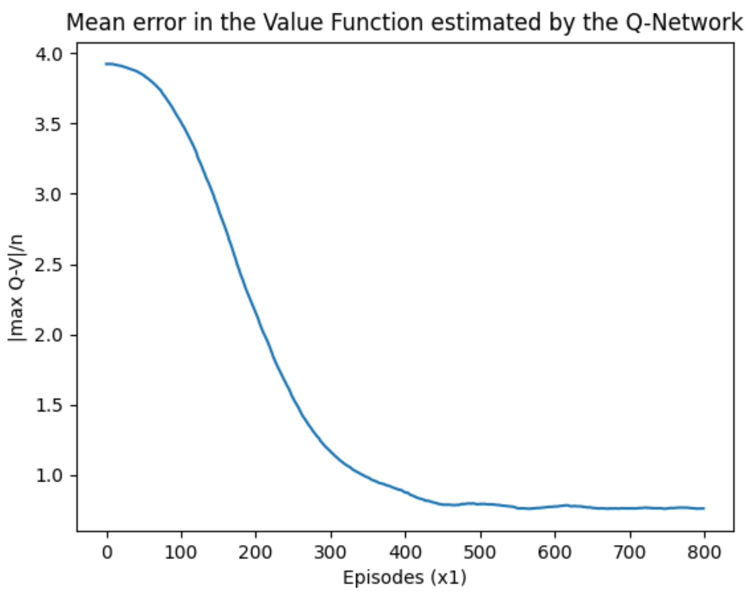
Figure 9 – Return at each training episode in the training process of a DQN for a small RL problem



Source: Own elaboration (2025)

This goes to show that a simple Q-learning algorithm is capable of approximating the Q function of the optimal policy in such a way that the true optimal path can be found. For this MDP with very small action and state spaces, this is clearly overkill, as value iteration can compute the optimal policy much more quickly; but as stated before, RL problems suffer from the "curse of dimensionality", in such a way that Q-learning and other approximate solution methods converge to a satisfactory result in a reasonable time, while the exact solution achieved through policy or value iteration takes way too long for RL problems with large state spaces and unimaginably long for continuous non-linear ones.

Figure 10 – Mean error committed by the DQN over training episodes in the training process



Source: Own elaboration (2025)

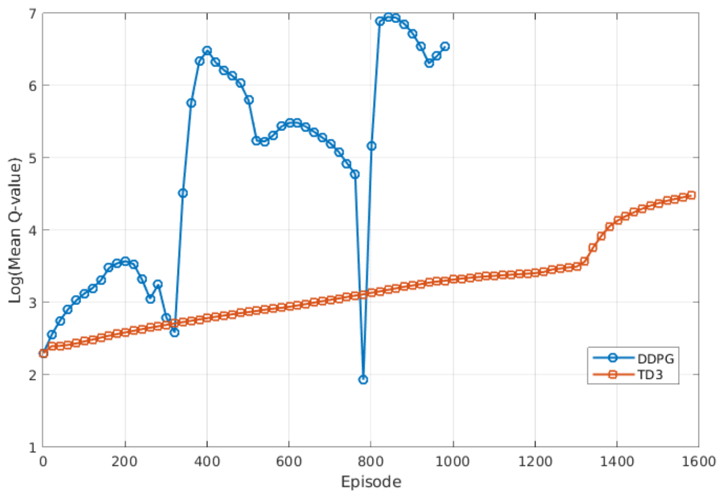
3.2 Learning Instability

In section 2.7.2, it was stated that the TD3 algorithm was made for addressing Q-value overestimation and learning instability problems. During the training loops for the cartpole system, such issues were encountered, and the way this was diagnosed was by monitoring the mean of the Q-values used for training (which, as reminder are chosen randomly from the memory buffer). TD3, however, was capable of learning high-performing policy without facing these instability issues, as can be seen in figure 11 (the logarithm of the Q values is plotted to illustrate better that the TD3 mean Q values were also growing).

This issue was first noticed by monitoring total rewards over episodes: the agent would seem to learn a better policy, then start getting worse rewards than before, as though it were "unlearning" what to do. Figure 12 shows this behavior occurring when a DQN was used to learn the stabilizing problem of the rotary pendulum, and figure 13 shows the mean Q-values computed by the DQN.

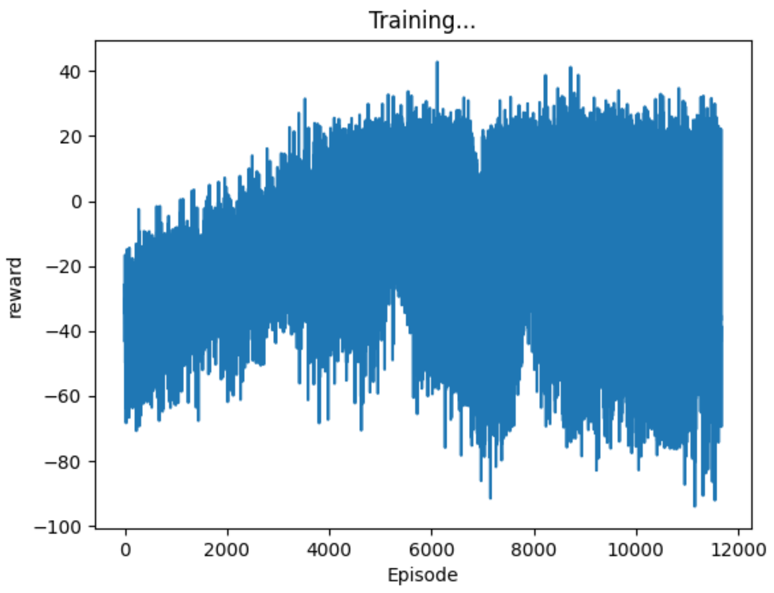
In the case of this DQN for the rotary pendulum, the issue was solved by lowering the target network's update rate τ from 0.005 to 0.001. Another way of avoiding this shortcoming is stopping the learning process when the agent is able to accumulate a certain threshold value of total rewards in a single run. This was done to obtain a sufficiently good policy for the comparisons in the following sections.

Figure 11 – Instability comparison of DDPG and TD3. The y-axis represents the log of the mean Q-values, and the x-axis represents the training episode.



Source: Own elaboration (2025)

Figure 12 – "Unlearning" behavior on a DQN for the stabilization problem of the rotary pendulum

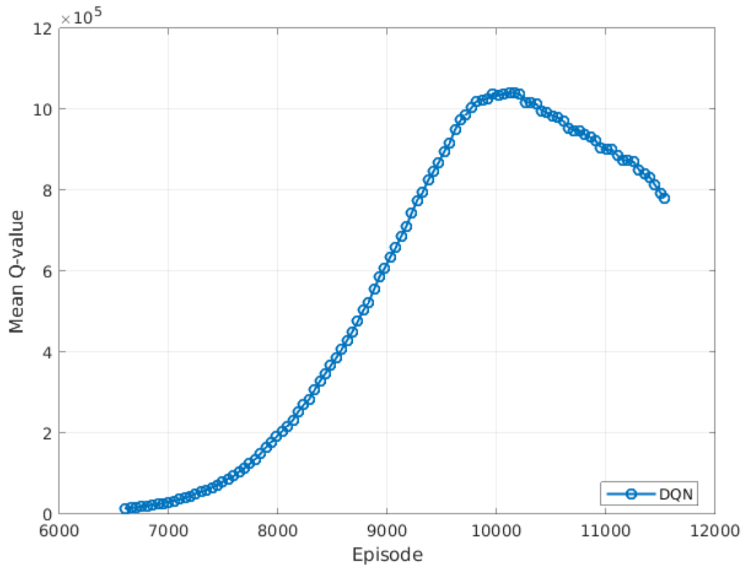


Source: Own elaboration (2025)

3.3 Controlling classic control systems near unstable equilibrium points

As stated in section 2.10, the three systems studied in this work are the simple pendulum, the cartpole, and the rotary pendulum, all of which are non-linear systems that, near any equilibrium point, can be controlled by a Linear Quadratic Regulator. Three LQRs were solved for the unstable equilibrium point of each of the three plants, and it was

Figure 13 – Mean Q-values of DQN for learning the stabilization problem of the rotary pendulum over episodes



Source: Own elaboration (2025)

observed that even if the initial conditions are not exactly $\vec{x} = \vec{0}$, the pendulum, cartpole, and rotary pendulum could be balanced by the LQR.

Since RL methods are capable of learning non-linear policies, it is expected that the farther away from the origin the initial conditions are, the better the RL agents will fare in comparison to the LQR solution. To test this, a DQN, a DDPG, and a TD3 agent were trained to balance each plant, each with two hidden layers containing 128 neurons. The test consists of running deterministic simulations with the LQR and the trained networks, varying the starting angle θ, and computing the total reward obtained across runs.

The following subsections detail the parameters of each system, the cost function that was chosen (which was, of course, the same for all agents and for computing the LQR's gain), and the results that were obtained.

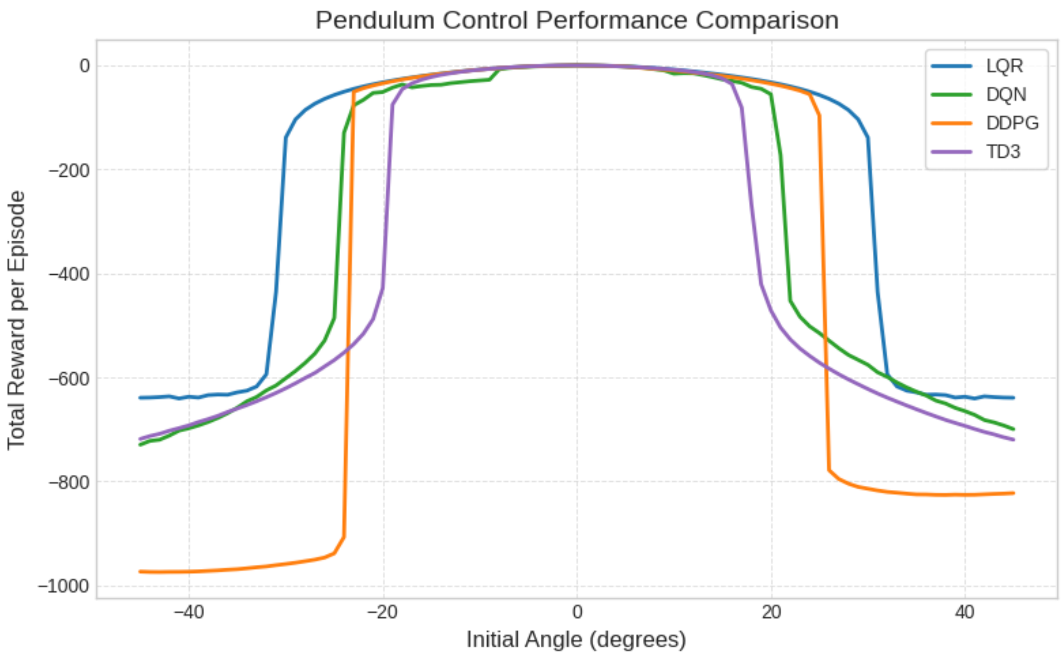
3.3.1 Simple Pendulum

The parameters of the pendulum described in section 2.10.1 are as follows: $m = 1.0\,\mathrm{kg}$, $l = 1.0\,\mathrm{m}$, and $b = 0.01\,\mathrm{N\,s\,m^{-1}}$ and the cost function is given in equation 3.1. The LQR's gain computed using an infinite horizon in MATLAB was $K = [-19.3141, 6.2241]$.

$$L(\theta, \dot{\theta}, u) = \theta^2 + 0.2\dot{\theta}^2 + 0.1u^2. \tag{3.1}$$

The maximum torque $\tau = u$ allowed for the control was the same $5\,\mathrm{N\,m}$, which was experimentally tested to be sufficient for the LQR. Regarding the quantization of the action space made for the DQN, the step was $0.5\,\mathrm{N\,m}$, such that the action space of the Q-learning agent was $\{-5.0, -4.5\ldots, 0, \ldots 4.5, 5.0\}$, and therefore the DQN had 21 output neurons. The starting angle was swept from $-45°$ to $+45°$ in steps of $0.5°$, and the results are given in figure 14.

Figure 14 – Total reward obtained after 100 time-steps by each RL agent and LQR over different starting angles θ_0 in the simple pendulum environment



Source: Own elaboration (2025)

As can be seen in the figure, no RL agent was capable of outperforming the LQR. DDPG was surprisingly more general and effective than DQN and TD3, but still could not control the pendulum for starting angles above 25°. The torque limitation likely played a significant role in this result, and the simplicity of the control problem explains the LQR's effectiveness.

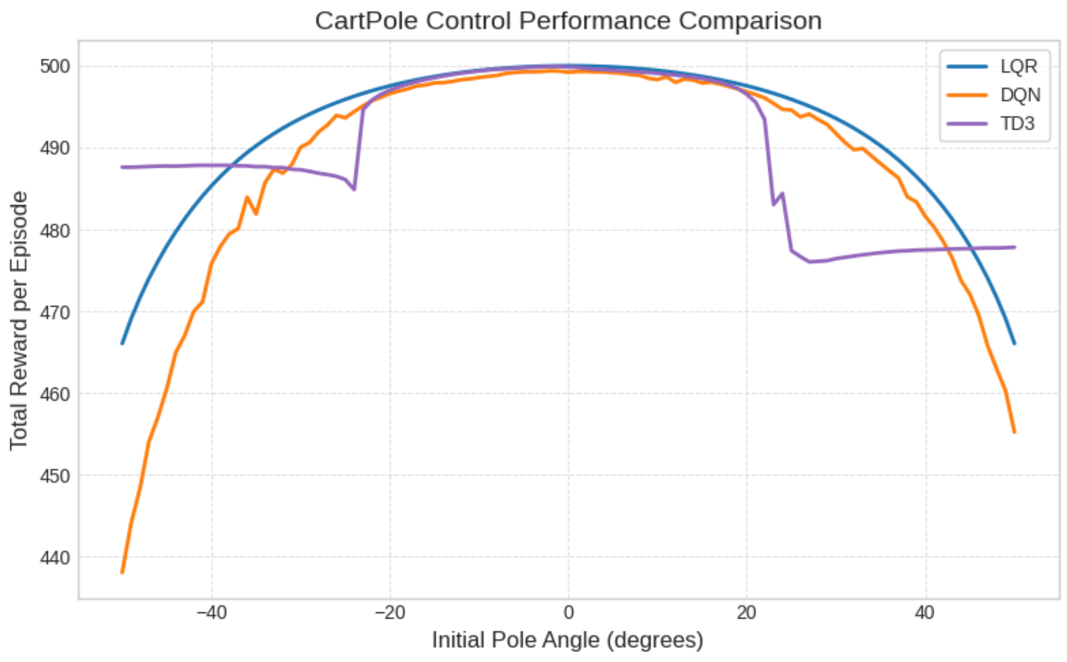
3.3.2 Cartpole

The system's parameters were the following: $m = 1.0\,\text{kg}$, $M = 2.0\,\text{kg}$, $l = 1.0\,\text{m}$, and the viscous friction b was set to zero. The maximum action signal was set to $F = u = 70\,\text{N}$, which is more than sufficient to stabilize the pendulum using an LQR (it was tested that starting at a 40° angle, the maximum force required was a bit bellow 65 N) and the quantization made for the DQN was in steps of 5 N, and therefore the output layer of the network had 27 neurons. Finally, the cost function of the problem is given in equation 3.2, which led to the following gain matrix for the LQR: $K = [-15.1744, -18.2369, -134.4547, -36.9102]$.

$$L(x, \dot{x}, \theta, \dot{\theta}, u) = 3.0x^2 + 0.1\dot{x}^2 + 2.0\theta^2 + 0.2\dot{\theta}^2 + 0.01u^2. \tag{3.2}$$

A positive offset was given to the cost function, which is why the total reward is positive in the plot; however, as stated in section 2.8, this constitutes a policy invariant transformation.

Figure 15 – Total reward obtained after 300 time-steps by TD3 and DQN agents and LQR over different starting angles θ_0 in the cartpole environment



Source: Own elaboration (2025)

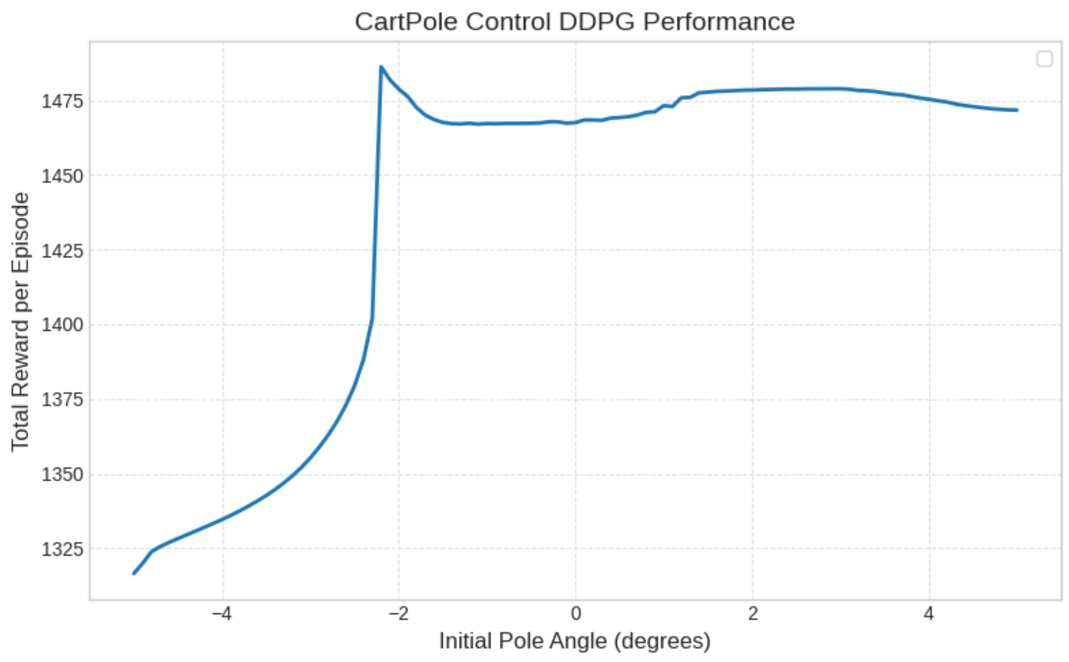
DDPG's performance on this task was disappointing. The agent suffered from the instability issues mentioned in section 3.2 even with a very low target network update rate (τ) and low learning rates. A suboptimal policy (capable of stabilizing the system with small initial angles) was saved to compare it with the TD3 and DQN agents; however, the rewards were still too low to fit neatly on the graph. For this reason, it is plotted separately in figure 16, with a more limited starting angle range (from $-5°$ to $+5°$), beyond which the reward went all the way down to -700000.

TD3, however, proved to have a very broad capability of successfully stabilizing the system with high starting angles at the cost of having a lower performance in the $22 - 39°$ range.

3.3.3 Rotary Pendulum

The rotary pendulum's physical parameters were the following: $m_p = 0.2\,\text{kg}$, $m_a = 0.01\,\text{kg}$, $M = 1.0\,\text{kg}$, $l_p = 1.0\,\text{m}$, $l_a = 0.5\,\text{m}$, and $I = 1.21 \times 10^{-3}\,\text{kg}\,\text{m}^2$. The maximum actuation signal allowed was $\tau = u = 20\,\text{N}\,\text{m}$, which again is sufficient for the LQR to stabilize the pendulum, and the quantization made for the DQN was in steps of

Figure 16 – Total reward obtained after 300 time-steps by DDPG agent in a limited angle range



Source: Own elaboration (2025)

0.5 N m, such that the output layer had 21 neurons. The LQR's gain for the cost function in equation 3.3 was $K = [-1.0739, -1.3304, -18.5823, -4.1347]$.

$$L(\phi, \dot{\phi}, \theta, \dot{\theta}, u) = 0.3\phi^2 + 0.03\dot{\phi}^2 + 3.0\theta^2 + 1.0\dot{\theta}^2 + 0.1u^2. \tag{3.3}$$

In this test case, all RL agents converged without instability, and TD3 achieved the best performance, matching LQR's rewards in the linear region and outperforming the other agents there. Furthermore, all RL agents had better performances than the LQR from about 18° onwards. The plot of the rewards can be seen in figure 17.

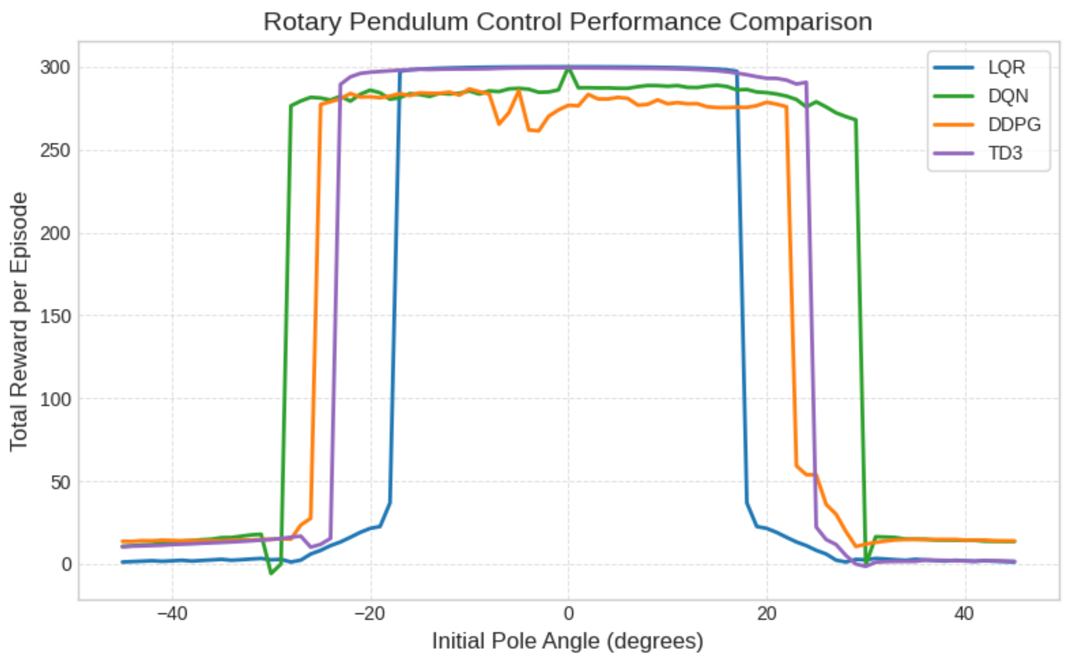
It appears that there is some trade-off between generality for non-linear control and performance in the linear region among the RL agents, as it is clear that the TD3, which outperforms the other agents in the linear region, had a narrower angle range in which it could balance the pendulum well. In contrast, the DDPG and DQN agents had inferior performances in the linear region and a broader range in which they could balance the pendulum.

3.4 Non-Linear tasks and Hybrid Control

3.4.1 Setup

As proposed in section 2.9, a hybrid control approach for the swing-up and stabilization tasks of all the aforementioned plants was utilized. This section compares the bare

Figure 17 – Total reward obtained after 300 time-steps by each RL agent and LQR over different starting angles θ_0 in the rotary pendulum environment



Source: Own elaboration (2025)

TD3 RL agents against the hybrid of LQR and TD3, using the control law of equation 2.15.

The physical parameters of the plants are all the same as in the other systems, except for the rotary pendulum, which proved itself to be very difficult for the RL algorithms to learn a way to swing the vertical arm upwards. That is because the determinant of the inertia matrix in 2.27 was low ($\det(D(0,0)) = 0.0197\,\mathrm{kg}^2\mathrm{m}^4$). Since it is still non-singular, the system is theoretically controllable and learning the task is feasible; however, it was decided to tweak the parameters in order to make the learning process, which still lasted 4000 episodes, somewhat shorter. The new inertia matrix had a determinant of $\det D(0,0) = 0.429\,\mathrm{kg}^2\mathrm{m}^4$ and its parameters are given in table 1

Table 1 – Rotary pendulum's parameters

Parameter	**Value**
m_p	$0.2\,\mathrm{kg}$
m_a	$0.01\,\mathrm{kg}$
M	$1.2\,\mathrm{kg}$
l_p	$2.0\,\mathrm{m}$
l_a	$1.2\,\mathrm{m}$
I	$1.21 \times 10^{-3}\,\mathrm{kg\,m}^2$

Source: Own elaboration (2025)

Furthermore, the rotary pendulum's cost function was also modified to $L(s,u) =$

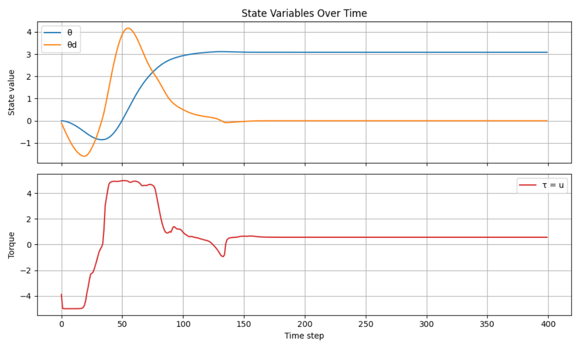
$0.3\phi^2 + 0.03\dot{\phi}^2 + 10.0\theta^2 + 1.0\dot{\theta}^2 + 0.0001u^2$, so as to incentivize the agent to seek to lower θ while not worrying too terribly much about the torque it would need to apply in order to do it and the LQR gain matrix that solves the control problem in the upright position for the new parameters and cost function was $K = [-3.33, -5.24, -43.40, -13.00]$. Finally, the TD3 agents employed to learn the task were always composed of neural networks with two hidden layers, each with 128 neurons, and a number of input and output neurons concordant to the the dimension of the observation and action space.

3.4.2 Reward and Evolution of the State Variables

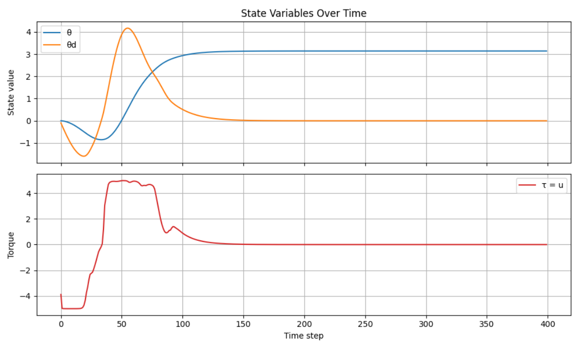
As shown in figures 18–20, the TD3 agent learned the swing-up task across all proposed systems. The total cost of all hybrid runs was lower than that of agents performing the swing-up and stabilization tasks on their own. However, the advantage was only marginal in the cartpole and simple pendulum systems. For the rotary pendulum, however, the hybrid approach was very effective, since the RL agent by itself had a very noisy control signal near the upright position. At the same time, that issue is eliminated when the LQR feedback gain takes control, as can be seen in figure 20b.

A quantitative comparison of hybrid control and pure RL is summarized in Table 2. There, the total cost accumulated in each run is displayed, and the relative improvement $\frac{\Delta J}{J_{\text{RL}}}$ is computed as a figure of merit for how much the hybrid control provides an advantage. It is notable that, even though the LQR control law contributes only to stabilization after the swing-up task is completed, there was a noticeable improvement in the performance metric. That improvement was most pronounced in the rotary pendulum system, which, as mentioned before, had a noticeable oscillation in the control signal when both tasks were left to the actor-critic.

Figure 18 – Evolution of the state variables in the swing-up task of the simple pendulum system.



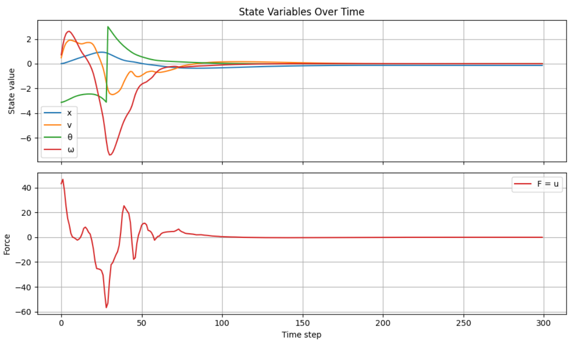
(a) TD3 RL agent.



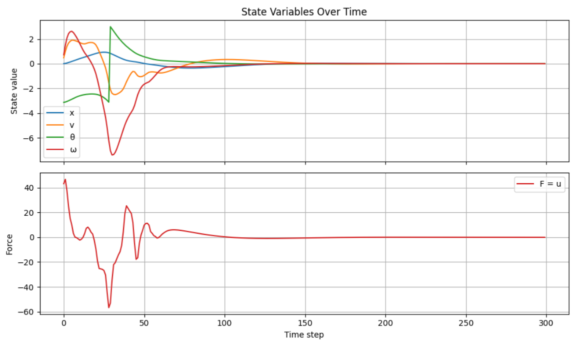
(b) Hybrid control.

Source: Own elaboration (2025)

Figure 19 – Evolution of the state variables in the swing-up task of the cartpole system.



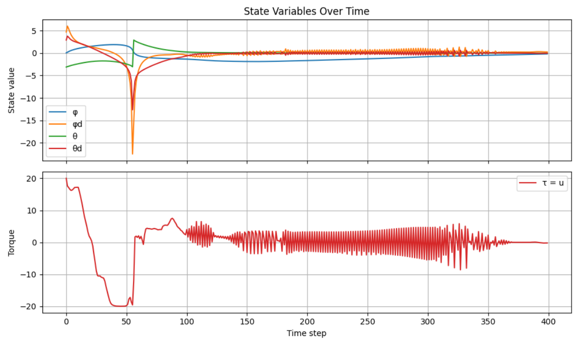
(a) TD3 RL agent.



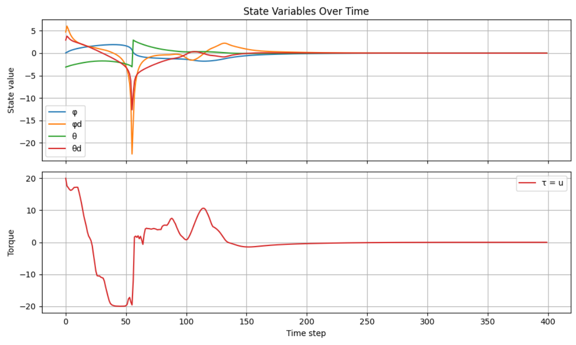
(b) Hybrid control.

Source: Own elaboration (2025)

Figure 20 – Evolution of the state variables in the swing-up task of the rotary pendulum system.



(a) TD3 RL agent.



(b) Hybrid control.

Source: Own elaboration (2025)

Table 2 – Overall cumulative cost obtained from the simulated runs of the proposed systems, using both the TD3 control law and the hybrid approach.

Systems	Pendulum		Cartpole		Rotary Pendulum	
Methods	pure RL	hybrid	pure RL	hybrid	pure RL	hybrid
Total Cost (a.u.)	744.82	734.66	1054.31	1038.36	4720.66	4605.70
Relative cost reduction (%)	1.36		1.51		2,44	

Source: Own elaboration (2025)

4 CONCLUSIONS

After laying the theoretical groundwork of RL and its connection to optimal control theory, breaking down each algorithm that was tested, and proposing a novel yet straightforward hybrid control method, we demonstrated the capability of Q-learning algorithms to learn the actual optimal policy in problems with small state spaces that can be solved through value iteration, even while the exact values computed by the network are not the same as those found through DP. This preliminary validation shows the extent to which approximate solution methods can match the exact solutions found through value iteration in an environment where they can be compared.

The RL algorithms implemented in this work were evaluated both for their performance in control tasks near equilibrium points and for their susceptibility to learning instability problems. Monitoring the mean Q-value computed by the critic networks was a handy tool for identifying Q-value overestimation and learning instability. TD3 and DQN proved to be the most stable alternatives compared to DDPG; the latter was satisfactory for problems with one-dimensional continuous action spaces, provided that the quantization step was sufficiently small. In contrast, the former was the most flexible and capable of finding a highly optimized policy, albeit at the cost of a longer training time.

Finally, the hybrid control scheme implemented proved very effective, particularly in the rotary pendulum system, which has non-linear dynamics stemming from both state variables. The method's advantage was very marginal in the other two systems; however, the results obtained still demonstrate the utility of this paradigm in control engineering, particularly for complex tasks with stabilization subtasks.

In summary, the results and contributions of this work demonstrate the relevance of reinforcement learning (RL) in control engineering, proving its viability in simulated environments. Planned future research in this field includes the design of a physical rotary pendulum with a DC motor, 3D-printed components, and an ESP32 controller, and the implementation of these tools for near-linear and non-linear control of this plant.

REFERENCES

BELLMAN, R. A markovian decision process. **Journal of mathematics and mechanics**, JSTOR, p. 679–684, 1957.

BROCKMAN, G. *et al.* Openai gym. **arXiv preprint arXiv:1606.01540**, 2016.

BRUNTON, S. L.; KUTZ, J. N. **Data-driven science and engineering: Machine learning, dynamical systems, and control**. [*S.l.: s.n.*]: Cambridge University Press, 2022.

FUJIMOTO, S.; HOOF, H.; MEGER, D. Addressing function approximation error in actor-critic methods. *In*: PMLR. **International conference on machine learning**. [*S.l.: s.n.*], 2018. p. 1587–1596.

GÄFVERT, M. Modelling the furuta pendulum. Department of Automatic Control, Lund Institute of Technology (LTH), 1998.

GHIO, A.; RAMOS, O. E. Q-learning-based model-free swing up control of an inverted pendulum. *In*: IEEE. **2019 IEEE XXVI International Conference on Electronics, Electrical Engineering and Computing (INTERCON)**. [*S.l.: s.n.*], 2019. p. 1–4.

HONGGE, R. *et al.* The balance control of two-wheeled robot based on bionic learning algorithm. *In*: IEEE. **The 26th Chinese Control and Decision Conference (2014 CCDC)**. [*S.l.: s.n.*], 2014. p. 4166–4170.

HORNIK, K.; STINCHCOMBE, M.; WHITE, H. Multilayer feedforward networks are universal approximators. **Neural networks**, Elsevier, v. 2, n. 5, p. 359–366, 1989.

KAELBLING, L. P.; LITTMAN, M. L.; MOORE, A. W. Reinforcement learning: A survey. **Journal of artificial intelligence research**, v. 4, p. 237–285, 1996.

LILLICRAP, T. P. *et al.* Continuous control with deep reinforcement learning. **arXiv preprint arXiv:1509.02971**, 2015.

MNIH, V. *et al.* Human-level control through deep reinforcement learning. **nature**, Nature Publishing Group, v. 518, n. 7540, p. 529–533, 2015.

NG, A. Y.; HARADA, D.; RUSSELL, S. Policy invariance under reward transformations: Theory and application to reward shaping. *In*: CITESEER. **Icml**. [*S.l.: s.n.*], 1999. v. 99, p. 278–287.

PENG, X. B. *et al.* Deepmimic: Example-guided deep reinforcement learning of physics-based character skills. **ACM Transactions On Graphics (TOG)**, ACM New York, NY, USA, v. 37, n. 4, p. 1–14, 2018.

PETERS, J.; VIJAYAKUMAR, S.; SCHAAL, S. Reinforcement learning for humanoid robotics. *In*: **Proceedings of the third IEEE-RAS international conference on humanoid robots**. [*S.l.: s.n.*], 2003. p. 1–20.

RANDLØV, J.; ALSTRØM, P. Learning to drive a bicycle using reinforcement learning and shaping. *In*: **ICML**. [*S.l.: s.n.*], 1998. v. 98, p. 463–471.

RICHTER, D. J.; CALIX, R. A.; KIM, K. A review of reinforcement learning for fixed-wing aircraft control tasks. **IEEE Access**, IEEE, 2024.

SILVER, D. *et al.* Mastering the game of go with deep neural networks and tree search. **nature**, Nature Publishing Group, v. 529, n. 7587, p. 484–489, 2016.

SUTTON, R. S.; BARTO, A. G. **Reinforcement Learning: An Introduction**. [*S.l.: s.n.*]: The MIT Press, 2018.

SUTTON, R. S. *et al.* Policy gradient methods for reinforcement learning with function approximation. **Advances in neural information processing systems**, v. 12, 1999.

TOWERS, M. *et al.* Gymnasium: A standard interface for reinforcement learning environments. **arXiv preprint arXiv:2407.17032**, 2024.

WANG, C. *et al.* Robust visuomotor control for humanoid loco-manipulation using hybrid reinforcement learning. **Biomimetics**, MDPI, v. 10, n. 7, p. 469, 2025.

WANG, D.; HUANG, H.; ZHAO, M. Model-free optimal tracking design with evolving control strategies via q-learning. **IEEE Transactions on Circuits and Systems II: Express Briefs**, IEEE, v. 71, n. 7, p. 3373–3377, 2024.

WANG, X. *et al.* Deep reinforcement learning: A survey. **IEEE Transactions on Neural Networks and Learning Systems**, IEEE, v. 35, n. 4, p. 5064–5078, 2022.

WATKINS, C. J.; DAYAN, P. Q-learning. **Machine learning**, Springer, v. 8, n. 3, p. 279–292, 1992.

WILLIAMS, R. J.; BAIRD, L. C. Tight performance bounds on greedy policies based on imperfect value functions. *In*: **In Proceedings of the Tenth Yale Workshop on Adaptive and Learning Systems**. [*S.l.: s.n.*], 1993.

WU, J. *et al.* A-td3: An adaptive asynchronous twin delayed deep deterministic for continuous action spaces. **IEEE Access**, IEEE, v. 10, p. 128077–128089, 2022.

ZEYNIVAND, A.; MOODI, H. Swing-up control of a double inverted pendulum by combination of q-learning and pid algorithms. *In*: IEEE. **2022 8th International Conference on Control, Instrumentation and Automation (ICCIA)**. [*S.l.: s.n.*], 2022. p. 1–5.