UNIVERSITA' POLITECNICA DELLE MARCHE

FACOLTA' DI INGEGNERIA

Corso di Laurea triennale Ingegneria Informatica e dell'Automazione

Reinforcement Learning per il controllo di sistemi meccatronici

Reinforcement Learning for mechatronic system control

Relatore: Chiar.mo/a

Prof. Emanuele Frontoni

Tesi di Laurea di:

Nicola Mochi

A.A. 2021 / 2022

INDEX

Chapter 1:

Introduction

What are the characteristics that make the human species so singular?

Our mind? Our capabilities to feel emotions? The ability to adapt ourselves to several kind of scenarios and habitats? Our amazing learning ability?

The idea that we learn by interacting with our environment is probably the first to arise to us when we think about the nature of learning. When an infant plays, waves its arms or looks about, it has no explicit teacher, but it does have a direct sensorimotor connection to its environment. Training this connection produces a treasure of information about cause and effect, the consequences of actions, and what to do in order to achieve goals. Throughout our lives, such interactions are undoubtedly a major source of knowledge about our environment and ourselves.

Whether we are learning to drive a car or master a new language, we are intensely aware of how our environment responds to what we do, and we seek to influence what happens through our behavior. Learning from interaction is a foundational idea underlying nearly all theories of learning and intelligence.

The main area of machine learning concerned with that theme is called Reinforcement Learning.

Reinforcement learning problems involve learning what to do, how to map situations to actions, to maximize a numerical reward signal. In an essential way they are closed-loop problems because the learning system's actions influence its later inputs. [1]

Reinforcement learning is a type of machine learning technique where a computer agent learns to perform a task through repeated trial and error interactions with an environment. This learning approach enables the agent to make a series of decisions that maximize a reward for the task without human intervention and without being explicitly programmed to achieve the task.

It is considered a branch of machine learning (see Figure 1). Unlike unsupervised and supervised machine learning, reinforcement learning does not rely on a static dataset, but operates in an environment and learns from collected experiences. Data points, or experiences, are collected during training through trial-and-error interactions between the environment and a software agent. This aspect of reinforcement learning is important, because it alleviates the need for data collection, preprocessing, and labeling before training, otherwise necessary in supervised and unsupervised learning. Practically, this means that, given the right incentive, a reinforcement learning model can start learning a behavior on its own, without (human) supervision.

Deep learning spans all three types of machine learning; Reinforcement learning and Deep learning are not mutually exclusive, as we will see. Complex reinforcement learning problems often rely on deep neural networks, a field known as deep reinforcement learning.

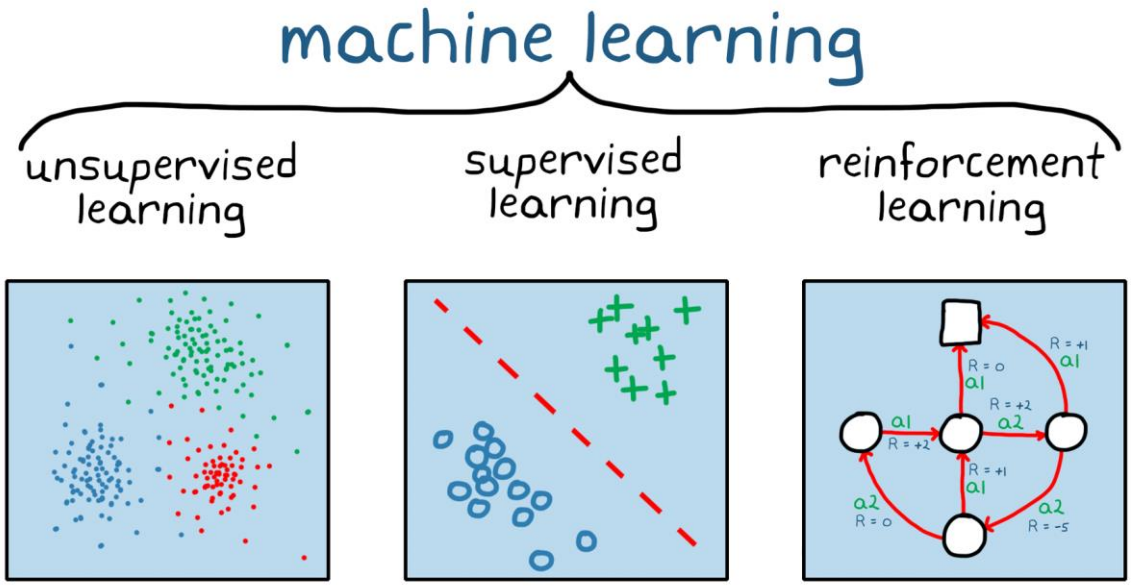


Figure 1. Categories of machine learning.

Deep neural networks trained with reinforcement learning can encode complex behaviors. This allows an alternative approach to applications that are otherwise intractable or more challenging to tackle with more traditional methods. For example, in autonomous driving, a neural network can replace the driver and decide how to turn the steering wheel by simultaneously looking at multiple sensors such as camera frames and lidar[1] measurements.

[1] Lidar is a method for determining ranges (variable distance) by targeting an object or a surface with a laser and measuring the time for the reflected light to return to the receiver

Without neural networks, the problem would normally be broken down into smaller pieces like extracting features from camera frames, filtering the lidar measurements, fusing the sensor outputs, and making “driving” decisions based on sensor inputs.

While reinforcement learning as an approach is still under evaluation for production systems, some industrial applications are good candidates for this technology.

Advanced controls: Controlling nonlinear systems is a challenging problem that is often addressed by linearizing the system at different operating points. Reinforcement learning can be applied directly to the nonlinear system.

Automated driving: Making driving decisions based on camera input is an area where reinforcement learning is suitable considering the success of deep neural networks in image applications.

Robotics: Reinforcement learning can help with applications like robotic grasping, such as teaching a robotic arm how to manipulate a variety of objects for pick-and-place applications. Other robotics applications include human-robot and robot-robot collaboration.

Scheduling: Scheduling problems appear in many scenarios including traffic light control and coordinating resources on the factory floor towards some objective. Reinforcement

learning is a good alternative to evolutionary methods to solve these combinatorial optimization problems.

Calibration: Applications that involve manual calibration of parameters, such as electronic control unit (ECU) calibration, may be good candidates for reinforcement learning.

As we said Reinforcement Learning, as one of the main characteristics of machine learning's fields, can be applied to a large variety of areas, from videogames and robotics to autonomous driving and statistics.

Although these different fields have a disparate level of complexity in applying Reinforcement Learning, the most complex ones are these involving the control of physical devices such as in mechanical or mechatronic systems, as the ones that we will meet in this work.

We are going to analyze the usage of Reinforcement Learning applied for controlling two different mechatronic systems.

Mechatronics is an engineering discipline integrating the fields of mechanics, electronics, control, and computer science. Many modern systems and products such as robots, manipulators, autonomous vehicles, electronic instruments, manufacturing equipment, and energy systems are designed and constructed by using mechatronic systems. The main characteristic of these systems is the progressively tighter coupling of mechanisms and an increasing number of electrical or electronic components with software. The arrangements

of these components and software ensure their functions, specifically in terms of their reliability, stability, and performance.

From the point of view of dynamics, mechatronic systems can be characterized by model uncertainties, high nonlinearities, complicated coupling, and stringent performance requirements, among others.

To deal with the abovementioned facts, strong research activity is still ongoing that aims to design efficient controllers that can guarantee high-performance control despite the presence of system uncertainties and external disturbances. Among the various control strategies, intelligent control (IC) methods with artificial intelligence have been widely studied and developed for mechatronic systems in terms of direct control, parameter optimization, system identification, uncertainty estimation, and compensation. [2]

The objective of this study is to investigate the potential of controllers based on Reinforcement Learning to see if it is possible to achieve better accuracy, robustness, reliability, and implementation simplicity.

Chapter 2:

State of the Art

Though being one of the youngest technologies, is rapidly growing the interest towards Reinforcement Learning.

Reinforcement learning as we previously said is a sub-domain of machine learning. It allows an agent to fulfil a given goal while maximizing a numerical reward signal. During the years it was developed within three main threads.

The first is the concept of learning by trial and error, discovered during researches undertaken in psychology and neuroscience of animal learning. The second concept is the problem of optimal control developed in the 1950s using a discrete stochastic version of the environment known as Markovian decision processes (MDP) and adopting a concept of a dynamical system's state and optimal return function (Reward) and defining the "Bellman equation" to optimize the agent behavior over the time. We will cover these topics in the later sections.

Moreover, alongside with RL we can use Deep Learning (DL) which allows us to improve the classic RL approach thanks to the concept of artificial neural networks that imitates human brain while processing data and creating patterns for use in decision-making. DL enables automatic features engineering and end-to-end learning through gradient descent and backpropagation. There are many types of DL nets, which usage depend on their application

and the nature of the problem being treated. For time sequences like speech recognition, natural language processing we use recurrent neural network. For extracting visual features, like image classification and object detection, we use convolutional neural network. For data pattern recognition like classification and segmentation, we use feed-forward networks, and for some complex tasks like video processing, object tracking and image captioning, we use a combination of those nets.

The link between RL and DL technologies was made, while AI researchers were seeking to implement a single agent that can think and act autonomously in the real world and get rid of any hand-engineered features. In fact, in 2015, Deepmind [2] succeed to combine RL, which is a decision-making framework and DL, which is a representation learning framework allowing visual features extraction, to create the first end-to-end artificial agent that achieves human-level performance in several and diverse domains. This new technology named deep reinforcement learning is used now, not only to play ATARI games, but also to design the next generation of intelligent self-driving cars like Google with Waymo, Uber, and Tesla. [3]

As stated in the Deepmind paper quoted above, learning to control agents directly from high-dimensional sensory inputs like vision and speech is one of the long-standing challenges of reinforcement learning (RL). Most successful RL applications that operate on these domains have relied on hand-crafted features combined with linear value functions or policy representations. The performance of such systems heavily relies on the quality of the feature representation. The advances in deep learning have made it possible to extract high-level

[2] DeepMind Technologies is a British artificial intelligence subsidiary of Alphabet Inc. and research laboratory founded in 2010. DeepMind was acquired by Google in 2014.

features from raw sensory data, leading to breakthroughs in computer vision and speech recognition. These methods utilize a range of neural network architectures.

It seems natural to ask whether similar techniques could also be beneficial for RL with sensory data. However, as the author says, reinforcement learning presents several challenges from a deep learning perspective. Firstly, most successful deep learning applications to date have required large amounts of hand labelled training data. RL algorithms, on the other hand, must be able to learn from a scalar reward signal that is frequently sparse, noisy, and delayed. The delay between actions and resulting rewards, which can be thousands of timesteps long, seems particularly daunting when compared to the direct association between inputs and targets found in supervised learning.

Another issue is that most deep learning algorithms assume the data samples to be independent, while in Reinforcement Learning, one typically encounters sequences of highly correlated states. Furthermore, in RL the data distribution changes as the algorithm learns new behaviors, which can be problematic for deep learning methods that assume a fixed underlying distribution. [4]

Another example of Reinforcement Learning overcoming humans is illustrated in another Deepmind paper that focuses on mastering chess.

The game of chess is the most widely studied domain in the history of artificial intelligence. The strongest programs are based on a combination of sophisticated search techniques, domain-specific adaptations, and handcrafted evaluation functions that have been refined by human experts over several decades. In contrast, the AlphaGo [3] Zero program recently

[3] AlphaGo is the first computer program, developed by Deepmind, to defeat a professional human Go player, the first to defeat a Go world champion.

achieved superhuman performance in the game of Go, by tabula rasa reinforcement learning from games of self-play. In the paper, they show how a single AlphaZero algorithm can achieve, tabula rasa, superhuman performance in many challenging domains. Starting from random play, and given no domain knowledge except the game rules, AlphaZero achieved within 24 hours a superhuman level of play in the games of chess and shogi (Japanese chess) as well as Go, and convincingly defeated a world-champion program in each case. [5]

As demonstrated by the latter papers, Reinforcement Learning fits perfectly with videogames like ATARI, chess, Go and so on.

More difficult is to adapt Reinforcement Learning algorithms as controllers.

Although is not impossible, and a lot of publications have been made about it, as the one of IEEE [4]about the adaptive control of a mechatronic system using RL.

In the article, the authors propose a simple and intuitive approach to improve the performance of a conventional controller in uncertain environments using deep reinforcement learning while maintaining safe operation. Their approach is motivated by the observation that conventional controllers in industrial motion control value robustness over adaptivity to deal with different operating conditions and are suboptimal as a consequence. Reinforcement learning, on the other hand, can optimize a control signal directly from input–output data and thus adapts to operational conditions but lacks safety guarantees, impeding its use in industrial environments. To realize adaptive control using reinforcement learning in such conditions, they follow a residual learning methodology, where a reinforcement

[4] The Institute of Electrical and Electronics Engineers (IEEE) is a professional association for electronic engineering and electrical engineering.

learning algorithm learns corrective adaptations to a base controller's output to increase optimality. [6]

Despite having achieved great and exciting results there are still some obstacles and open questions in which research is trying to deepen.

Among all, we can mention sample efficiency, exploration vs. exploitation, reward shaping, and hyperparameters tuning.

When we talk about sample efficiency, we are actually talking about the number of interactions required for an agent to learn a good model of the environment.

For example, in the DeepMind paper quoted before by Hessel et al., they showed that in order to reach human-level performance on an Atari game running at 60 frames per second they needed to run 18 million frames, which corresponds to 83 hours of play experience. For an Atari game that most humans pick up within a few minutes, this is a lot of time. So, in this case we can say that RL is sample inefficient.

Exploration vs exploitation is a fundamental trade-off for RL algorithms. When our agent has to choose the action to take, it faces a central issue: either exploit the information collected so far to make the currently best decision or explore for more information. Balance correctly these two features is one of the biggest challenges in Reinforcement Learning. [7]

The Reward Function is an incentive mechanism that tells the agent what is correct and what is wrong using reward and punishment. The goal of agents in RL is to maximize the total rewards.

Reward shaping is a big deal. If you have sparse rewards, you don't get rewarded very often. If our robotic arm is only going to get rewarded when it stacks the blocks successfully, then

all the time it's off exploring far away it will never get any feedback. This takes a much longer time.

You want to instead shape rewards that get gradual feedback and let it know it's getting better and getting closer. It helps it to learn faster.

When we use neural networks in Deep Reinforcement Learning we have a huge amount of what are called hyperparameters. Hyperparameters are parameters whose values control the learning process and determine the values of model parameters that a learning algorithm ends up learning.

However, setting the right hyperparameters can have a huge impact on the deployed solution performance and reliability in the inference models, produced via RL, used for decision-making. Hyperparameter search itself is a laborious process that requires many iterations and is computationally expensive to find the best settings that produce the best neural network architectures.

Reinforcement Learning is one of the most exciting prospects in the field of AI, and it's gaining popularity not only in the academic research, but also in big companies such as Google, Facebook, Amazon, Microsoft, IBM.

2.1 The Company: Siemens

Siemens is a German multinational conglomerate corporation and the largest industrial manufacturing company in Europe headquartered in Munich with branch offices abroad.

The principal divisions of the corporation are Industry, Energy, Healthcare and Infrastructure & Cities, which represent the main activities of the corporation.

Following the 2018 restructuring plan, starting from fiscal 2019, the Siemens group is divided into the three operating companies Digital Industries, Smart Infrastructure, Gas and Power, and into the three strategic companies Siemens Mobility, Siemens Healthineers and Siemens Gamesa Renewable Energy.

Siemens Digital Industries is a company founded by Siemens AG in 2019 following the merger of its two previous divisions Digital Factory and Process Industries and Drives.

The company is headquartered in Nuremberg, Germany, and has approximately 78,000 employees worldwide. The products and services offered concern the digitization and automation of production processes, such as, for example, software for factories and software for product life cycle control, machines, sensors, systems for mechatronics, cloud platforms and for the internet of things.

I had the honour of spending three months in the branch of Siemens Industry Software based in Leuven, Belgium.

Siemens Industry Software NV (SISW LMS) is a Belgian company, constituting the Simulation and Test Solutions (STS) business segment within Siemens PLM Software.

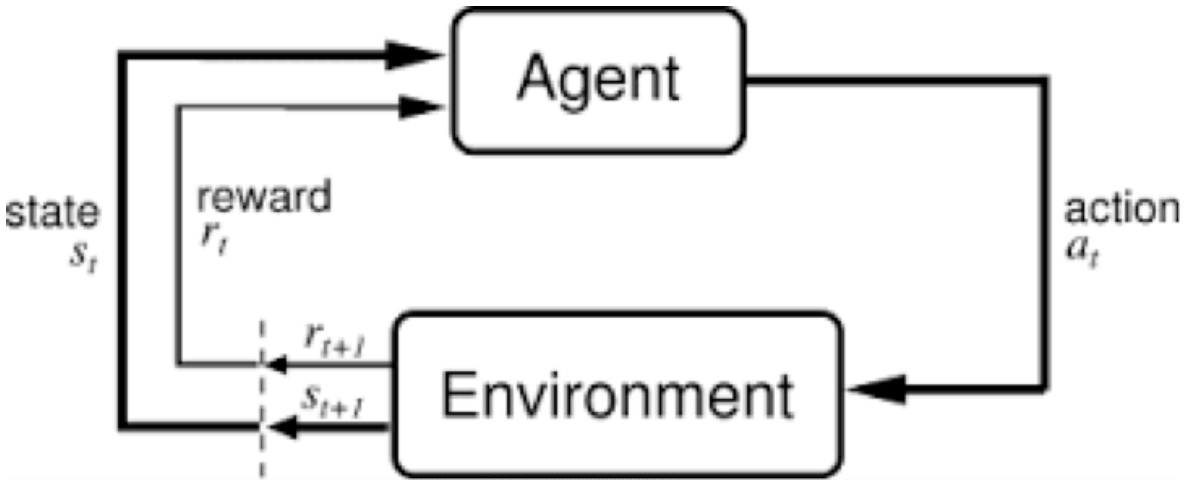
SISW originated as LMS International, a spin-off company from the KU Leuven, developing into a world-leading position for advanced engineering solutions for the mechanical and mechatronic industry. In 2013, LMS was acquired by Siemens to complement its offering on design engineering software within the ”Digital Factory” division. More specifically, SISW LMS develops and markets testing systems, multi-domain simulation software and engineering services to support product design in relation to critical performance parameters such as safety, fuel economy, noise and vibration, structural integrity and lifetime. SISW LMS is headquartered in Leuven, Belgium, with a staff of about 350 and has R&D centres throughout Europe, the US and India. While its main markets are the automotive and aerospace ones, its solutions are adopted in all sectors where mission critical performance must be ensured. SISW LMS continuously innovates its solutions through substantial investments in R&D.

Part of these investments is dedicated to seeking new, alternative and attractive, solutions alongside with the classic ones, as Reinforcement Learning.

Chapter 3:

Theoretical Background

3.1 Basics of Reinforcement Learning



The main characters of RL are the agent and the environment. The environment is the world in which the agent lives and interacts with. At every step of interaction, the agent sees a (possibly partial) observation of the state of the world, and then decides on an action to take. The environment changes when the agent acts on it, but may also change on its own.

The agent also perceives a reward signal from the environment, a number that tells it how good or bad the current world state is. The goal of the agent is to maximize its cumulative reward, called return. Reinforcement learning methods are ways that the agent can learn behaviors to achieve its goal.

To talk more specifically what RL does, we need to introduce additional terminology. We need to talk about:

- states and observations,
- action spaces,
- policies,
- trajectories,
- rewards,
- value functions.

3.1.1 States and Observations

A state "s" is a complete description of the state of the world we are considering, while an observation "o" is a partial description of a state which may omit some information.

In Deep Reinforcement Learning we use to represent states and observations by a vector, matrix, or a higher-order tensor.

When the agent is able to observe the complete state of the environment, we say that the environment is fully observed. When the agent can only see a partial observation, we say that the environment is partially observed[5].

[5] Sometimes the notation 's' is used where it should be technically more appropriate to write the symbol for observation, 'o'.

3.1.2 Action spaces

The set of all valid actions in a given environment is often called the action space.

Action space is strictly connected with the environment, in fact different environments allow different action spaces, they could be very different.

For example, when we use RL in videogames, like Atari or Go, we have discrete action spaces, while for an agent which controls a robot in the physical world and in other real-life applications, we operate in continuous action spaces.

This difference causes distinctions in RL algorithms' family, in fact some of them could be applied only in one case.

3.1.3 Policies

The policy in Reinforcement Learning, often denoted with **μ** defines what action, from the defined action space, the agent should take at a certain state in the environment.

The policy is the core of a reinforcement learning agent in the sense that it alone is sufficient to determine behavior.

The policy can be categorized as:

- **Deterministic policy:** there is only one specific action possible in each given state. When the agent reaches a given state, the deterministic policy tells it to perform a certain action always.
- **Stochastic Policy:** it returns a probability distribution of multiple actions in the action space for a given state.

Most of the times the policies are stochastic.

In deep RL, we deal with parameterized policies: policies whose outputs are computable functions that depend on a set of parameters (e.g., the weights and biases of a neural network) which we can adjust to change the behavior via some optimization algorithm.

3.1.4 Trajectories

A trajectory $\boldsymbol{\tau}$ is a sequence of states and actions in the world,

$$\tau = (\mathrm{s}_{0,}, a_0, s_1, a_1) \quad (1)$$

state transitions (what happens to the world between the state at time t, s_t and the state at t_{+1}, $s_{t+1,}$) are governed by the natural laws of the environment, and depend on only the most recent action, a_t. They can be either deterministic, or stochastic.

Trajectories are often also called episodes or rollouts.

3.1.5 Rewards

The reward function R is critically important in reinforcement learning. It depends on the current state of the world, the action just taken, and the next state of the world:

$$r_t = \mathrm{R}\ (\ \mathrm{s}_{\mathrm{t},}, a_t, s_{t+1}\ ) \quad (2)$$

The goal of the agent is to maximize some notion of cumulative reward over a trajectory.

One kind of return is the finite-horizon undiscounted return, which is just the sum of rewards obtained in a fixed window of steps:

$$R = \sum_{t=0}^{T} r_t \quad (3)$$

Another kind of return is the infinite-horizon discounted return, which is the sum of all rewards ever obtained by the agent, but discounted by how far off in the future they're obtained. This formulation of reward includes a discount factor $\gamma \in (0,1)$.

$$R = \sum_{t=0}^{\infty} \gamma^t r_t \quad (4)$$

We insert a discount factor because under reasonable conditions, the infinite sum can converge.

3.1.6 Value Functions

It's often useful to know the value of a state, or state-action pair. By value, we mean the expected return if you start in that state or state-action pair, and then act according to a particular policy forever after. Value functions are used, one way or another, in almost every RL algorithm.

Two of the most important ones are:

- The Optimal Value Function, V*(s), which gives the expected return if you start in state s and always act according to the optimal policy in the environment:

$$V^*(s) = \max_{\pi} E\,[\,R \mid s_0 = s\,] \quad (5)$$

- The Optimal Action-Value Function, Q*(s, a), which gives the expected return if you start in state s, take an arbitrary action a, and then forever after act according to the optimal policy in the environment:

$$Q^*(s, a) = \max_{\pi} E\,[\,R \mid s_0 = s,\ a_0 = a\,] \tag{6}$$

There is a strict connection between the optimal action-value function Q*(s, a) and the action selected by the optimal policy. In fact, Q*(s, a) gives the expected return for starting in state s and taking (arbitrary) action a. The optimal policy in s will select whichever action maximizes the expected return from starting in s. As a result, if we have Q*, we can directly obtain the optimal action, a*(s), with:

$$a^*(s) = arg\ \max_{a} Q*(s, a) \tag{7}$$

3.1.7 Exploration vs Exploitation

As we said, one of the challenges that arise in reinforcement learning, and not in other kinds of learning, is the trade-off between exploration and exploitation. To obtain a lot of rewards, a reinforcement learning agent must prefer actions that it has tried in the past and found to be effective in producing rewards. But to discover such actions, it has to try actions that it has not selected before.

The agent has to exploit what it already knows in order to obtain rewards, but it also has to explore in order to make better action selections in the future. The dilemma is that neither exploration nor exploitation can be pursued exclusively without failing at the task. The agent must try a variety of actions and progressively favour those that appear to be best. On a stochastic task, each action must be tried many times to gain a reliable estimate its expected reward. [8]

3.2 Algorithms

One of the most important branching points in an RL algorithm is the question of whether the agent has access to (or learns) a model of the environment. By a model of the environment, we mean a function which predicts state transitions and rewards.

The main upside to having a model is that it allows the agent to plan by thinking ahead, seeing what would happen for a range of possible choices, and explicitly deciding between its options. Agents can then distill the results from planning ahead into a learned policy. A particularly famous example of this approach is AlphaZero. When this works, it can result in a substantial improvement in sample efficiency over methods that don't have a model.

The main downside is that a ground-truth model of the environment is usually not available to the agent. If an agent wants to use a model in this case, it has to learn the model purely from experience, which creates several challenges. The biggest challenge is that bias in the model can be exploited by the agent, resulting in an agent which performs well with respect to the learned model, but behaves sub-optimally (or super terribly) in the real environment. Model-learning is fundamentally hard, so even intense effort—being willing to throw lots of time and compute at it—can fail to pay off.

Algorithms which use a model are called Model-based methods, and those that don't are called Model-free. While Model-free methods forego the potential gains in sample efficiency from using a model, they tend to be easier to implement and tune.

We will mainly focus on Model-free algorithms which in turn, can be approached in two ways: **Policy Optimization** and **Q-Learning.**

Methods in Policy Optimization family represent a policy explicitly as $\boldsymbol{\pi_\theta(a|s)}$. They optimize the parameters $\boldsymbol{\theta}$ either directly by gradient ascent on the performance objective $J(\pi_\theta)$, or indirectly, by maximizing local approximations of $J(\pi_\theta)$. This optimization is almost always performed on-policy, which means that each update only uses data collected while acting according to the most recent version of the policy. Policy optimization also usually involves learning an approximator $V_\phi(s)$ for the on-policy value function $V^\pi(s)$, which gets used in figuring out how to update the policy.

On the other hand, methods in the family of Q-learning learn an approximator $Q_\theta(s,a)$ for the optimal action-value function, $Q^*(s,a)$. Typically they use an objective function based on the Bellman equation. This optimization is almost always performed off-policy, which means that each update can use data collected at any point during training, regardless of how the agent was choosing to explore the environment when the data was obtained. The corresponding policy is obtained via the connection between Q^* and π^*: the actions taken by the Q-learning agent are given by [9]

$$a(s) = \arg\max_a Q_\theta(s,a) \tag{8}$$

We are going to show a simple example to better explain how Q-learning works.

Let's suppose that we are playing a board game (represented in the top-left corner), in which we want the car to reach the trophy.

Game Board:

Current state (*s*): 0 0 0
0 1 0

Q Table: γ = 0.95

	0 0 0 1 0 0	0 0 0 0 1 0	0 0 0 0 0 1	1 0 0 0 0 0	0 1 0 0 0 0	0 0 1 0 0 0
⇧	0.2	0.3	1.0	-0.22	-0.3	0.0
⇩	-0.5	-0.4	-0.2	-0.04	-0.02	0.0
⇨	0.21	0.4	-0.3	0.5	1.0	0.0
⇦	-0.6	-0.1	-0.1	-0.31	-0.01	0.0

We can represent all the possible states in columns (1 stands for the actual position of the car) and possible actions in rows, in what's called Q-table.

Q-table helps us to find the best action for each state. After each time step, the Q value for that specific action is updated according to some reward signal.

The reward signal can be discounted by some value between 0 and 1 (γ).

Basically, this table will guide us to the best action at each state.

So, we will select the action which leads to the maximum Q value in that column:

Game Board:

Current state (s): 000 010

Selected action (a): ⇨

Q Table: γ = 0.95

	000 100	000 010	000 001	100 000	010 000	001 000
⇧	0.2	0.3	1.0	-0.22	-0.3	0.0
⇩	-0.5	-0.4	-0.2	-0.04	-0.02	0.0
⇨	0.21	0.4	-0.3	0.5	1.0	0.0
⇦	-0.6	-0.1	-0.1	-0.31	-0.01	0.0

At the beginning the reward is 0. Now we are right under the trophy (this can be noticed by the position of the 1 in the third column):

Game Board:

Current state (s): 000 010

Selected action (a): ⇨

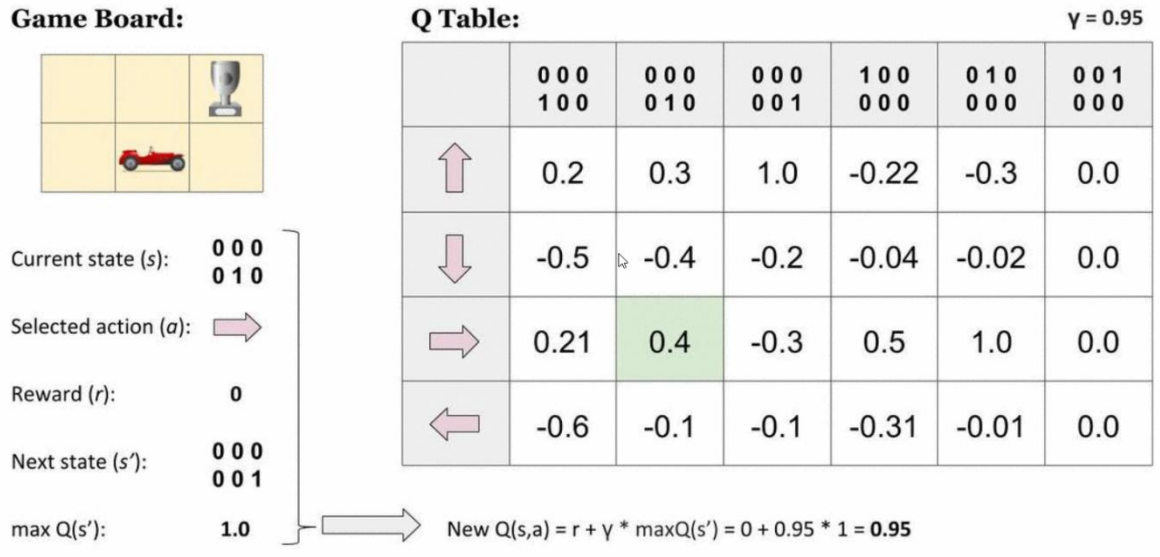
Reward (r): 0

Next state (s'): 000 001

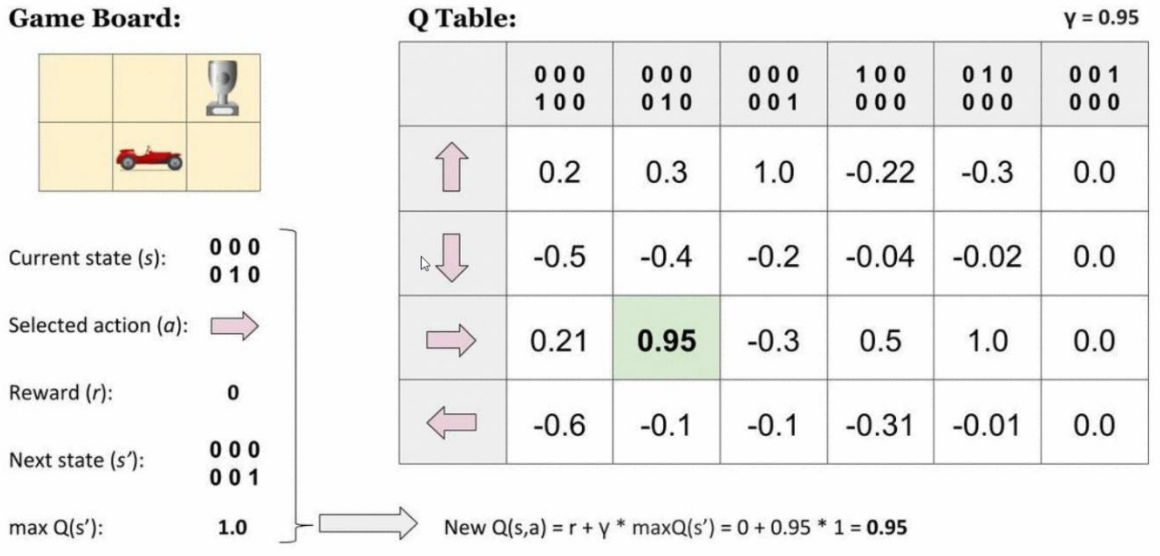
Q Table: γ = 0.95

	000 100	000 010	000 001	100 000	010 000	001 000
⇧	0.2	0.3	1.0	-0.22	-0.3	0.0
⇩	-0.5	-0.4	-0.2	-0.04	-0.02	0.0
⇨	0.21	0.4	-0.3	0.5	1.0	0.0
⇦	-0.6	-0.1	-0.1	-0.31	-0.01	0.0

The highest Q value that we can get is by picking the action ⇧ in order to reach the trophy (Q value of 1.0).



Hence, the Q value will be updated:



So, we could set our algorithm environment as a grid (if we are playing a game e.g.) as we just saw.

However, while it's manageable to create and use a Q-table for simple environments, it's quite difficult with some real-life environments. In fact, the number of actions and states in a real-life environment can be thousands, making it extremely inefficient to manage q-values in a table. So, most of the times we use neural networks, more powerful, from which Deep Q-learning derives.

Deep learning neural networks, or artificial neural networks, attempts to mimic the human brain through a combination of data inputs, weights, and bias. These elements work together to accurately recognize, classify, and describe objects within the data.

Deep neural networks consist of multiple layers of interconnected nodes, each building upon the previous layer to refine and optimize the prediction or categorization. This progression of computations through the network is called forward propagation. The input and output layers of a deep neural network are called visible layers. The input layer is where the deep learning model ingests the data for processing, and the output layer is where the final prediction or classification is made.

Another process called backpropagation uses algorithms, like gradient descent, to calculate errors in predictions and then adjusts the weights and biases of the function by moving backwards through the layers in an effort to train the model. [10]

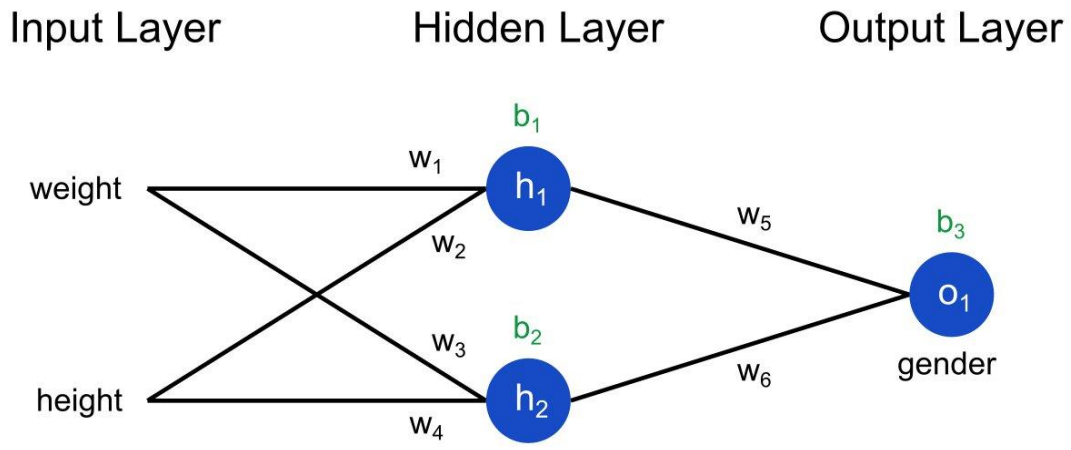


Figure 2. Basic example of a neural network [11]

Together, forward propagation and backpropagation allow a neural network to make predictions and correct for any errors accordingly. Over time, the algorithm becomes gradually more accurate.

There is a wide range of choice when talking about Reinforcement Learning Algorithms and at the time of writing a lot of them are being discovered.

Nevertheless, for the purpose of our study we will serve of two of them: Deep Q-learning (DQN) and Deep Deterministic Policy Gradient (DDPG).

3.2.1 Deep Q-Learning (DQN)

Deep Q Learning uses the Q-learning idea and takes it one step further. Instead of using a Q-table, we use a Neural Network that takes a state and approximates the Q-values for each action based on that state.

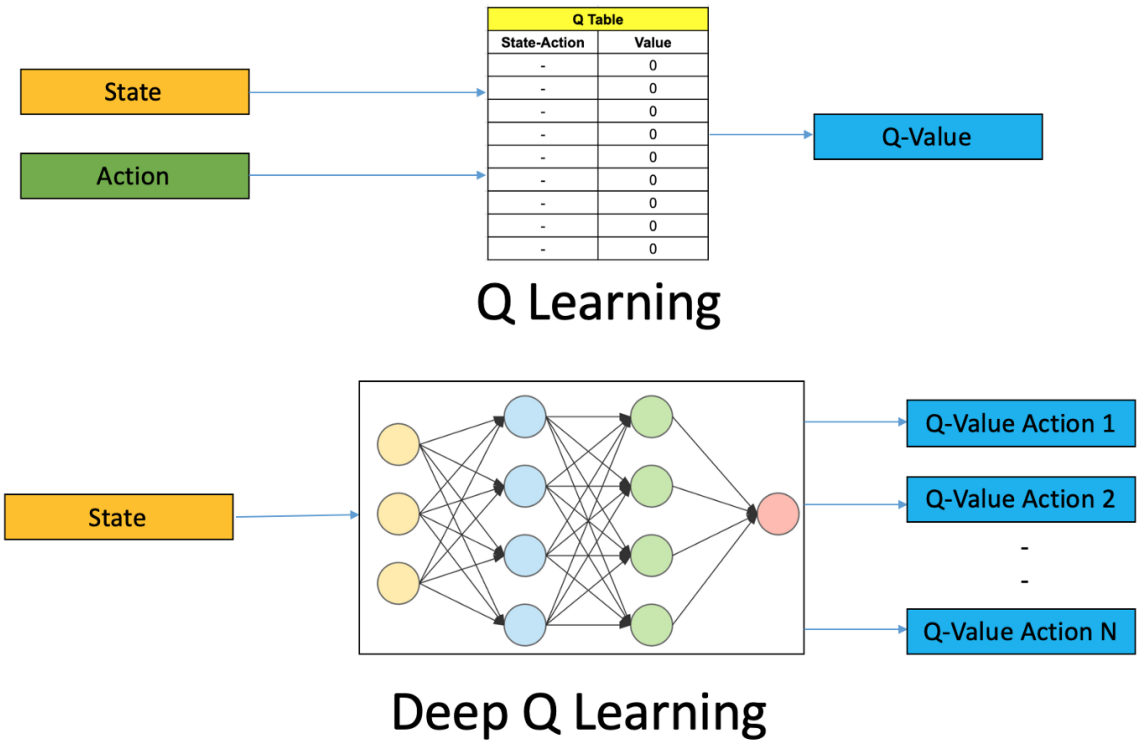


Figure 3. Difference between Q-Learning and Deep Q-Learning.

In DQN is utilized a technique known as experience replay where the agent's experiences is stored at each time-step, $e_t = (s_t, a_t, r_t, s_{t+1})$ in a data-set $D = e_1, \ldots, e_N$ pooled over many episodes into a replay memory. During the inner loop of the algorithm, Q-learning updates are applied, or minibatch updates, to samples of experience, $e \sim D$, drawn at random from the pool of stored samples. After performing experience replay, the agent selects and executes an action according to an ε-greedy policy. Since using histories of arbitrary length

as inputs to a neural network can be difficult, Q-function instead works on fixed length representation of histories produced by a function φ. The full algorithm, developed by the authors of the paper, which is called deep Q-learning, is presented in the following image.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Second, learning directly from consecutive samples is inefficient, due to the strong correlations between the samples; randomizing the samples breaks these correlations and therefore reduces the variance of the updates. Third, when learning on-policy the current parameters determine the next data sample that the parameters are trained on. For example, if the maximizing action is to move left then the training samples will be dominated by samples from the left-hand side; if the maximizing action then switches to the right then the training distribution will also switch. It is easy to see how unwanted feedback loops may arise and the parameters could get stuck in a poor local minimum, or even diverge catastrophically. By using experience replay the behavior distribution is averaged over many of its previous states, smoothing out learning and avoiding oscillations or divergence in the parameters. When learning by experience replay, it is necessary to learn off-policy (because

our current parameters are different to those used to generate the sample), which motivates the choice of Q-learning.

In practice, the latter algorithm only stores the last N experience tuples in the replay memory, and samples uniformly at random from D when performing updates. This approach is in some respects limited since the memory buffer does not differentiate important transitions and always overwrites with recent transitions due to the finite memory size N. Similarly, the uniform sampling gives equal importance to all transitions in the replay memory. A more sophisticated sampling strategy might emphasize transitions from which we can learn the most. [4]

In conclusion, another key feature of DQN algorithm is that they can be used only within discrete action spaces. That derives from the fact that the selection of actions is based on the action-value function, so in most cases they are only suitable for discrete action spaces.

So, if we want to apply it in a scenario where the action space is continuous, we have to discretize it.

3.2.2 Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradient (DDPG) is an algorithm which concurrently learns a Q-function and a policy. It uses off-policy data and the Bellman equation to learn the Q-function and uses the Q-function to learn the policy.

This approach is closely connected to Q-learning and is motivated the same way: if you know the optimal action-value function $Q^*(s, a)$, then in any given state, the optimal action $a^*(s)$ can be found by solving the equation (7).

DDPG interleaves learning an approximator to $Q^*(s, a)$ with learning an approximator to $a^*(s)$, and it does so in a way which is specifically adapted for environments with continuous action spaces. But what does it mean that DDPG is adapted specifically for environments with continuous action spaces? It relates to how we compute the max over actions in

$$\max_a Q^*(s, a).$$

When there are a finite number of discrete actions, the max poses no problem, because we can just compute the Q-values for each action separately and directly compare them. (This also immediately gives us the action which maximizes the Q-value.) But when the action space is continuous, we can't exhaustively evaluate the space, and solving the optimization problem is highly non-trivial. Using a normal optimization algorithm would make calculating $\max_a Q^*(s, a)$ a painfully expensive subroutine. And since it would need to be run every time the agent wants to take an action in the environment, this is unacceptable. Because the action space is continuous, the function $Q^*(s, a)$ is presumed to be differentiable with respect to the action argument. This allows us to set up an efficient, gradient-based learning rule for a policy $\mu(s)$ which exploits that fact. Then, instead of running an expensive optimization subroutine each time we wish to compute , we can approximate it with [12]

$$\max_a Q(s, a) \approx Q(s, \mu(s)).$$

Deep Deterministic Policy Gradient algorithms fall into the area of Actor-Critic Methods. Actor–critic methods are TD methods that have a separate memory structure to explicitly represent the policy independent of the value function. The policy structure is known as the actor, because it is used to select actions, and the estimated value function is known as the critic, because it criticizes the actions made by the actor. Learning is always on-policy: the critic must learn about and critique whatever policy is currently being followed by the actor. The critique takes the form of a TD error. This scalar signal is the sole output of the critic and drives all learning in both actor and critic, as suggested by Figure 4.

Typically, the critic is a state-value function. After each action selection, the critic evaluates the new state to determine whether things have gone better or worse than expected. [1]

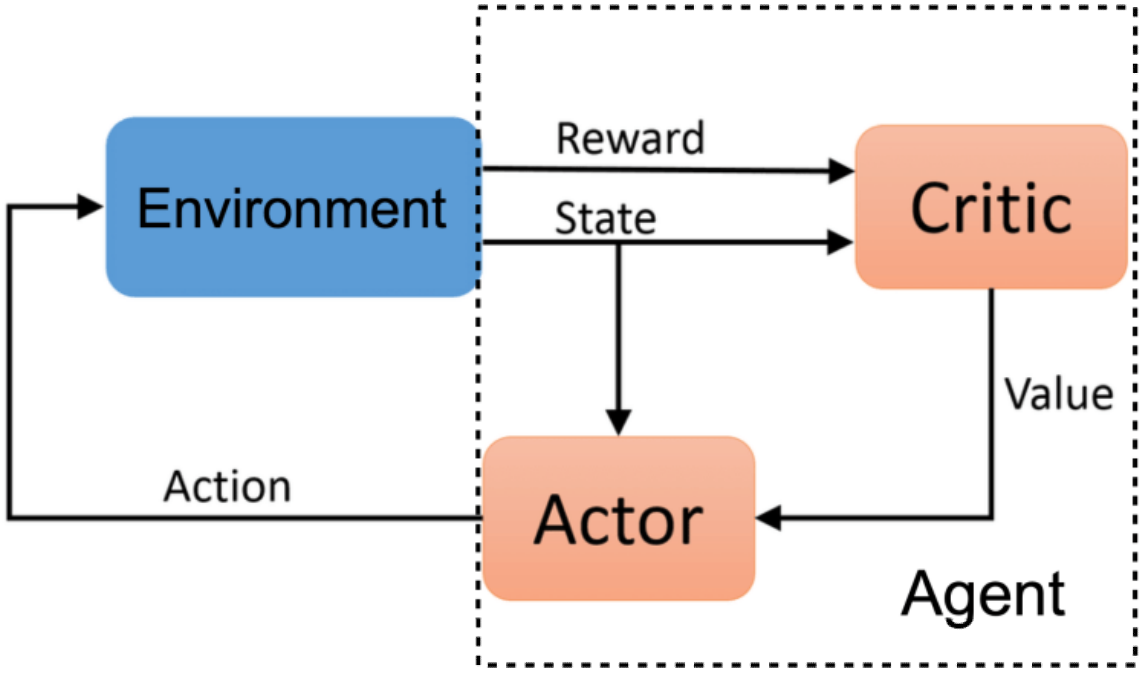


Figure 4. Actor-Critic methods.

In the below image is showed the pseudocode of the algorithm.

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t 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 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:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Chapter 4:

Results

In this chapter we are going to show the results of the usage of Reinforcement Learning algorithms for the control of mechatronic system.

Especially we will go through two different mechatronic systems which we will try to control using the Reinforcement Learning algorithms that we showed before: DQN and DDPG.

We decided to focus on these ones because we wanted to start with algorithm quite simple to tune, but at the same time very powerful.

Before starting to show the models of the systems we are going to briefly introduce the main Siemens software that were used for this study: Simcenter Studio and Simcenter Amesim.

Simcenter Amesim is the leading integrated, scalable system simulation platform, allowing system simulation engineers to virtually assess and optimize the performance of mechatronic systems. Simcenter Amesim combines ready-to-use multi-physics libraries with the application-and industry-oriented solutions that are supported by powerful platform capabilities, to let you rapidly create models and accurately perform analysis. This open environment can be easily coupled with major computer-aided engineering (CAE), computer-aided design (CAD), and controls software packages.

However, we used Simcenter Amesim only to access the model of the mechatronic systems that we studied (without giving data about the model to the algorithms because we used Model- Free algorithms).

We developed the algorithms in Simcenter Studio.

Simcenter Studio is an application in the Simcenter portfolio for generating and evaluating system architectures during the early concept phase. The software includes patented technology for engineers and data scientists to create novel and topologically different system architectures. Simcenter Studio also combines system simulation, optimal control methods, and reinforcement learning on top of a state-of-the-art machine learning and scientific computing stack to automatically simulate and evaluate hundreds of these architectures. This allows engineers and data scientists to create user defined procedures in computational notebooks for generative engineering.

Computational notebooks combine narrative text, mathematical equations, code, models, and visualization so that workflows, user-defined procedures, executable models, and documentation are included in the same place.

We built our algorithms with computational notebooks based on scriptable interface using Python.

4.1 First use case: Tubing Array

In the following pages will be showed the first system proposed to us for our study denominated Tubing Array.

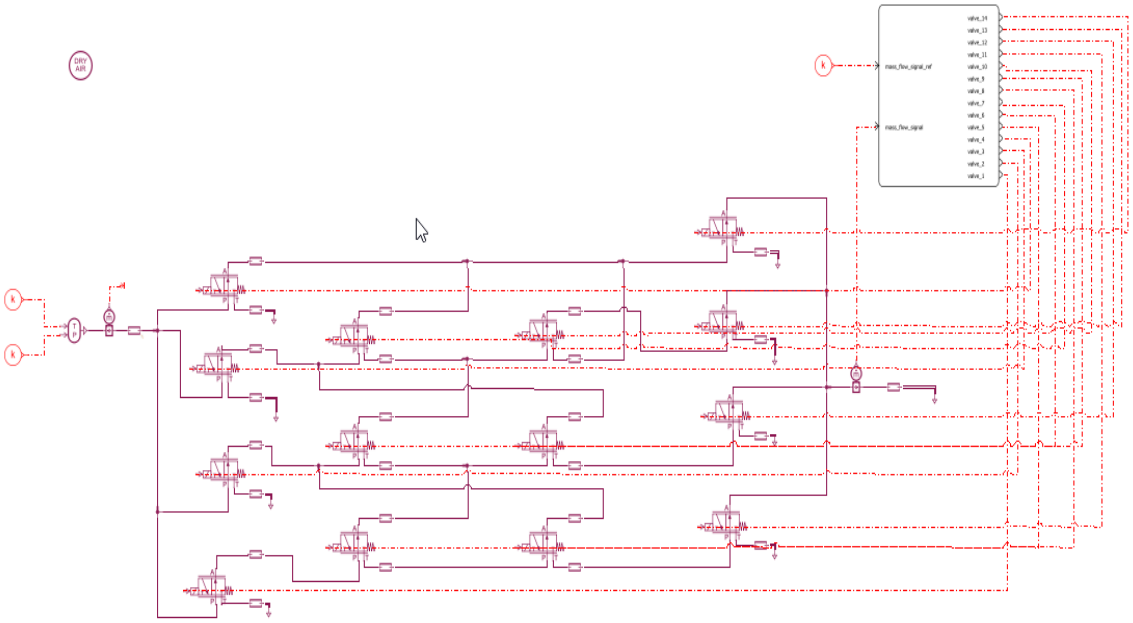


Figure 5. First mechatronic system: Tubing Array

The latter one is the model of the mechatronic system built in Amesim.

The system consists of a net of 14 pneumatic valves connected to each other in 4 layers.

The system, which controls the gas flux in the valves, is composed as follow:

- 2 main inputs: constant value of temperature and pressure, represented by the two ‘k’ at the beginning.

- Mass flow signal: it’s the output produced by valve’s network, whose signal goes as an input for the interface.

- The Interface: it's the top-right block in Figure 5. It is the component which allows us to connect our Amesim's model with Simcenter Studio.

- Mass flow signal reference: a reference of the mass flow signal defined by us which is the second input to the interface.

Each valve can have an input signal of 0 or 1. We can assign 0 for the valve closed and 1 for the valve open.

All the parameters derived from the Amesim model, both the ones about temperature and pressure and the ones characteristic of the valves, were provided to us.

Our goal is to implement a Reinforcement Learning algorithm, whose role will be controlling each valve's input so that we will have the mass flow signal (output) as close as possible to the reference defined.

To develop our algorithms we will use, as we said, the computational notebooks provided by Simcenter Studio using the programming language Python.

Simcenter Studio provides us a large number of tools and built-in function to develop Reinforcement Learning algorithms easily.

4.1.1 DQN

We are going to develop our algorithm and apply it to control the mechatronic system showed before.

For this purpose, we will use Simcenter Studio with which we can connect to the model defined in Amesim thanks to the interface.

So, we will have to control 14 inputs signal to produce an output as close as possible to the reference. As we are using DQN we have to discretize the action space. To do so we can use a built-in function that allows us to discretize the action space by giving as an input all the combinations of possible actions.

Since we want to firstly see if this algorithm could fit our system and work well with it, we will start by reducing the number of inputs from 14 to 4. By doing this it will be easier to tune the algorithm.

To do so we will consider the inputs of only 4 out of the 14 valves and we will assume the other ones constantly closed to 0.

Before showing the results of the algorithm we show below the reward function that we used for the algorithm.

```
abs_error = abs(o_tp1[1] - o_tp1[0])
r_tp1 = 0.01/(abs_error+0.01)
```

Figure 6. Reward function for DQN algorithm

We decided to start with a simple reward function.

In the first line of Figure 6 we instantiate in the variable called "abs_error" the absolute value of the difference between the mass flow signal and its reference.

Having this difference, we want to give a reward to the agent inversely proportionate to the gap between the output of the system and its reference. This because our goal is to try to find an algorithm which can learn the input configuration of the valves' actions that minimize the gap between system's output and the reference.

So, we can guess that the smaller the "abs_error" variable will be the better will be for our system. Furthermore, being the reward inversely proportional to the error, the bigger the error will be the smaller the reward will be.

The value of 0.01 is justified by the fact that we can't simply give a reward of $\frac{1}{abs\_error}$ because otherwise when the value of the error is 0 (ideal case or in the first iteration when the value of the output and the reference are set to 0) the reward would be infinite.

Moreover, we gave an additional value of 0.01 in order to not give a reward too large for the agent. We will present and then discuss the results of our studies by observing some graphs created with Simcenter Studio.

In fact, Studio is so powerful that allows us to create with simple built-in function graphs related to:

- the Accumulated rewards during episodes;
- the trend of mass flow signal (output) in comparison with its reference, during the timesteps of an episode. This will represent the state space during an episode;

- the value of the actions picked by the agent through the timesteps of an episode, which will represent the action space during an episode.

All the graphs that we are going to show will be related to the training and the testing of our algorithm.

In RL, training (also known as learning) generally refers to the use of RL algorithms, such as Q-learning, to estimate the optimal policy.

The test phase instead, it used to evaluate the policy learned by the algorithm during training time.

The evaluation phase of an RL algorithm is the assessment of the quality of the learned policy and how much reward the agent obtains if it follows that policy.

Training:

We made use of early stopping of the training useful to stop the training of a Reinforcement learning agent when the reward reaches a certain value.

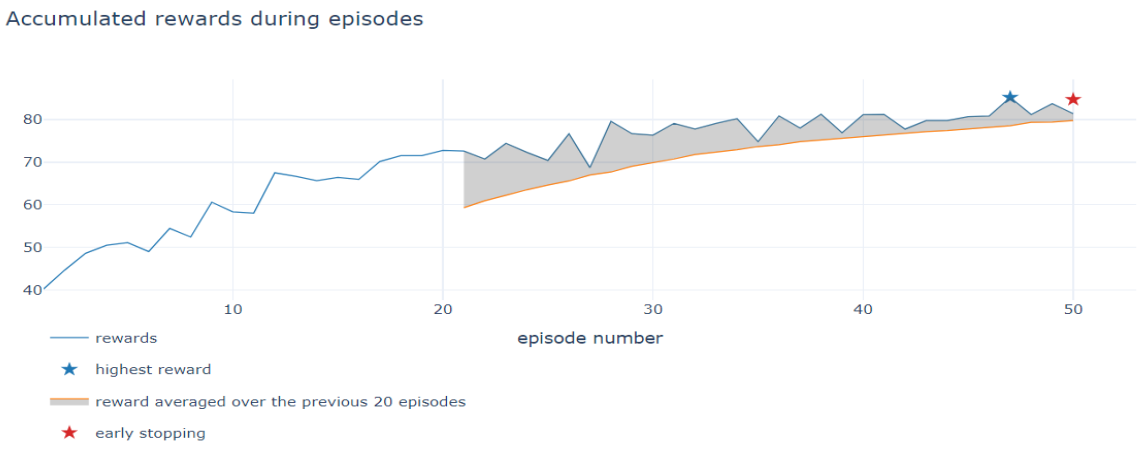


Figure 7. Training of 100 episodes of 200 timesteps. Highest reward of 84.7 reached at episode 48.

This graph regards the trend of the output of the system (mass flow signal, orange line) and the reference that the system should follow (blue line), during the best episode, the episode which led to the highest reward, of the training.

The value of the reference in this configuration and in this system is set to 10.5[6] .

We can see that after few timesteps the agent manages to follow quite well the reference, despite having some "turbulence": in the signal.

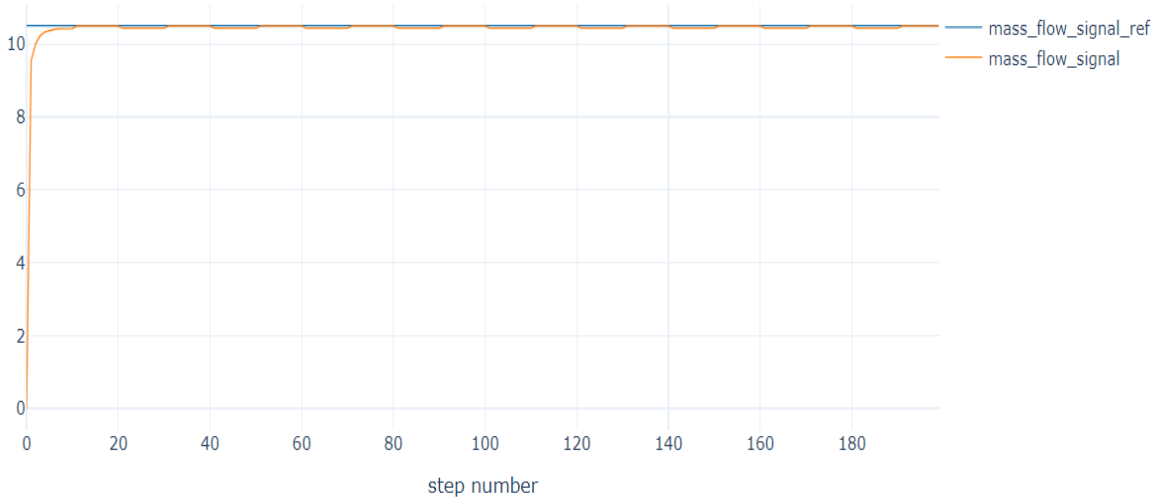


Figure 8. ***Training:*** *Best episode's state space.*

[6] This decision was made because we didn't have any clues about what should have been the reference value. For this reason, we ran a simulation with Simcenter Amesim (without the interface), which showed us that without any variation the system was producing values of the mass flow signal fluctuating around the value of 10.5, hence the reason of the choice.

The image below instead regards the action space during the best episode, which represents the value of the actions picked by the agent. It learns that by choosing the action in that way, it can reach higher rewards.

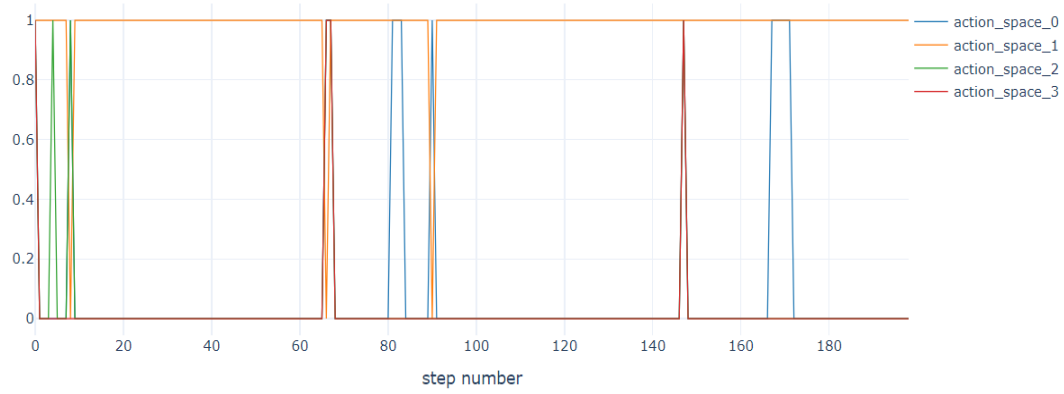
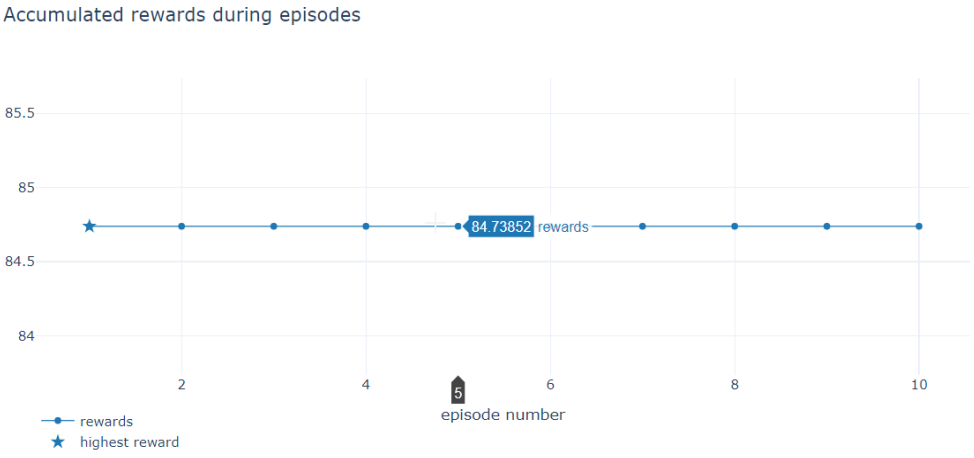


Figure 9. ***Training:*** *Best episode's action space.*

Since by the training the policy adopted seems quite good, we are going to evaluate it through testing.

Testing

As we said training regards assessment of the quality of the learned policy, so the accumulated rewards during episodes will be of the same quantity during the episode as we can see from the picture:



Regarding the state space it is very similar to the best one of the training, still being not perfectly constant.

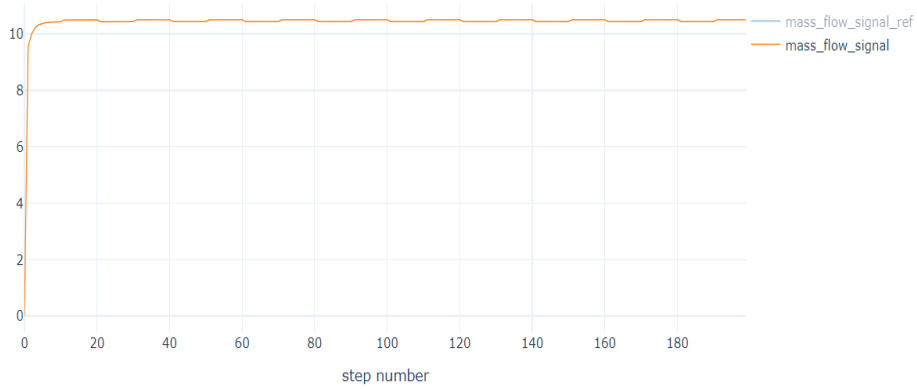


Figure 10. ***Testing:*** *state space.*

Interesting is to talk about the action space. Here we can see that, differently from the training episode, the agent choses as best actions to keep 3 valves at an input signal of 0 (closed) and the other one at 1 (valve open). This could be interesting because it could suggest a possible configuration in order to reach that output value.

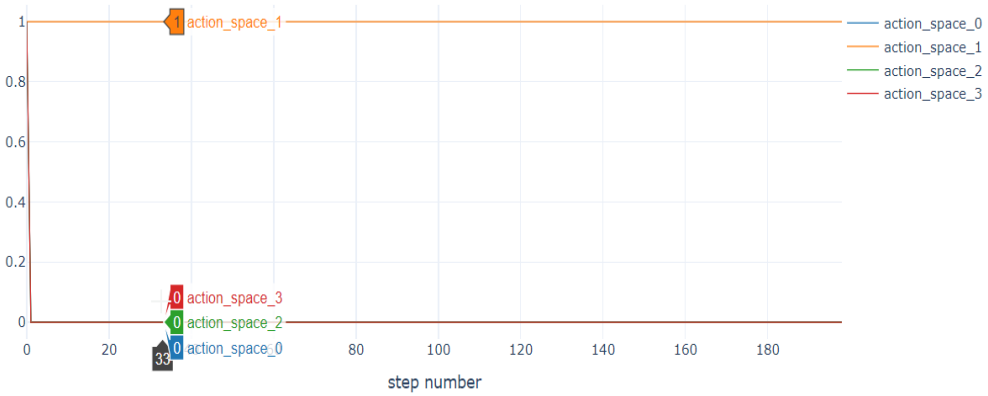


Figure 11. ***Testing:*** *action space.*

Nevertheless, is difficult to see and give an objective meaning to the graphs that we saw above, so we will help us by using different measures of the error, of the gap between the output and the reference.

We will use MSE (Mean Squared Error), MAE (Mean Absolute Error) and the sum of the total error during the episode.

In statistics, the mean squared error (MSE) measures the average of the squares of the errors, that is the average squared difference between the estimated values Y_i and the actual value $\hat{Y}_1$.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

Advantage: The MSE is great for ensuring that our trained agent has no outlier predictions with huge errors, since the MSE puts larger weight on these errors due to the squaring part of the function.

Disadvantage: If our agent makes a single very bad measurement, the squaring part of the function magnifies the error.

On the other hand, MAE is calculated as the sum of absolute errors divided by the sample size:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|Y_i - \hat{Y}_i|$$

Advantage: MAE directly covers the MSE disadvantage. Since we are taking the absolute value, all of the errors will be weighted on the same linear scale. Thus, unlike the MSE, we won't be putting too much weight on our outliers and our loss function provides a generic and even measure of how well our agent is performing.

Disadvantage: If we do in fact care about the outlier measurements of our agent, then the MAE won't be as effective. The large errors coming from the outliers end up being weighted the exact same as lower errors. This might result in our agent being great most of the time, but making a few very poor predictions every so-often. [13]

For that system we had the following values.

MSE	MAE	Total error
0.56	0.094	18.96

These are the parameters used with DQN algorithm.

Buffer size	1 000 000
Discount rate	0.9
Initial exploration rate	0.9
Decay exploration rate	0.065
Minimum exploration rate	0.2
Learning rate	0.0005

Table 1. DQN's parameters

Now we can move into the full, initial system of 14 valves.

As said at the beginning DQN is an algorithm that operates in discreate action spaces, meaning that again we have to discretize since we are working on a continuous environment.

We could simply do that with the simple structure of 4 valves, because in the code implementation, we could easily insert all actions' combinations (in total 16).

Nevertheless, with the 14 valves system we should provide 2^{14} = 16384 combinations of actions.

That would mean that at each iteration our agent should choose among them, leading to very long, resource-intensive trainings, which would require a huge amount of memory usage, and that could require a lot of time in order to produce an interesting solution for our objective. So, in conclusion is not useful to implement a DQN algorithm for that type of system, hence we will choose another one.

4.1.2 DDPG

In line with the latter consideration, we will implement another algorithm for our agent, DDPG.

As we said in chapter 3 DDPG operates in a continuous action space, hence we don't have to discretize it.

We will follow the same operating method adopted for the previous algorithm, so we will start with the 'reduced' system of 4 valves.

Training

The red stars indicate the value of the reward of the early stopping, evaluated every 50 episodes.

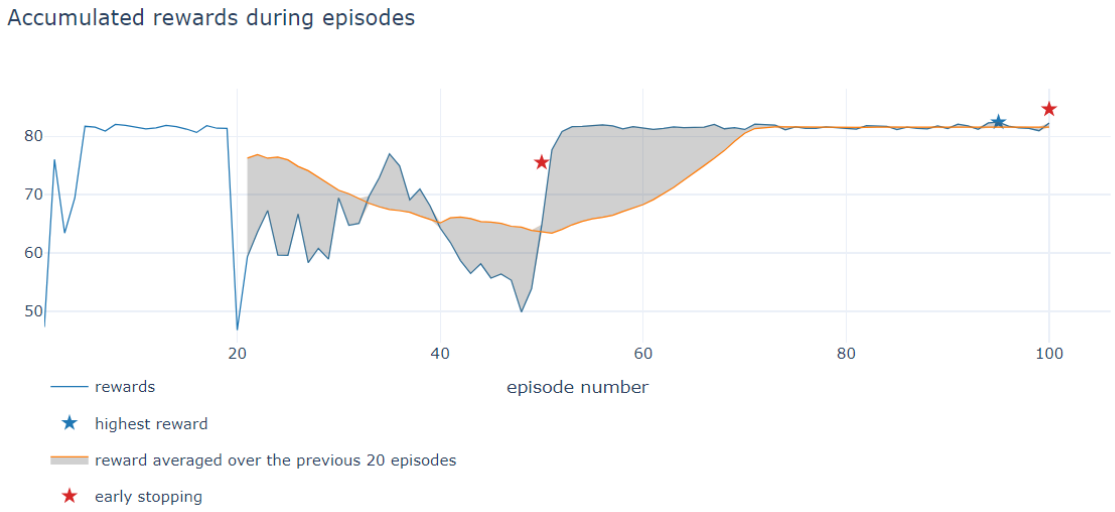


Figure 12. Training of 100 episodes of 200 timesteps. Best episode: 96. Total reward 84.6

The trend here is quite strange, the agent seems to have found the best policy within the first episodes but then is acting as if it's trying to find a better one without success. Despite all, after 50 episodes it manages to find again the initial trend that seems to be the best one.

Also, the amount of reward is similar to the one we got with DQN with the same system. Regarding the state space, again, the behavior is very similar to DQN's one.

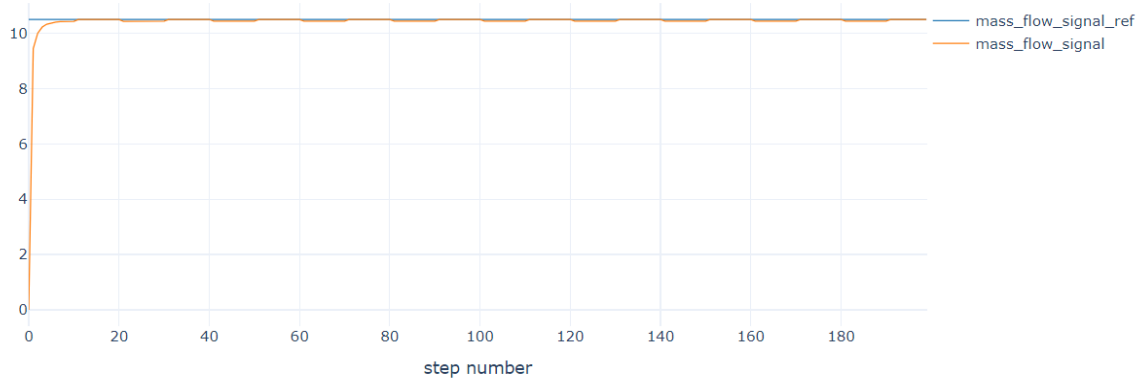


Figure 13. ***Training:*** *Best episode's state space.*

The action space is a bit different. In fact, with DDPG we can define the range of the action space from which our agent can pick its actions, hence we set it between 0 and 1. However DDPG can pick continuous action. It does so [7] with only one valve, whose values despite are not so far from 1, while the other 3 are still to 0, as with DQN.

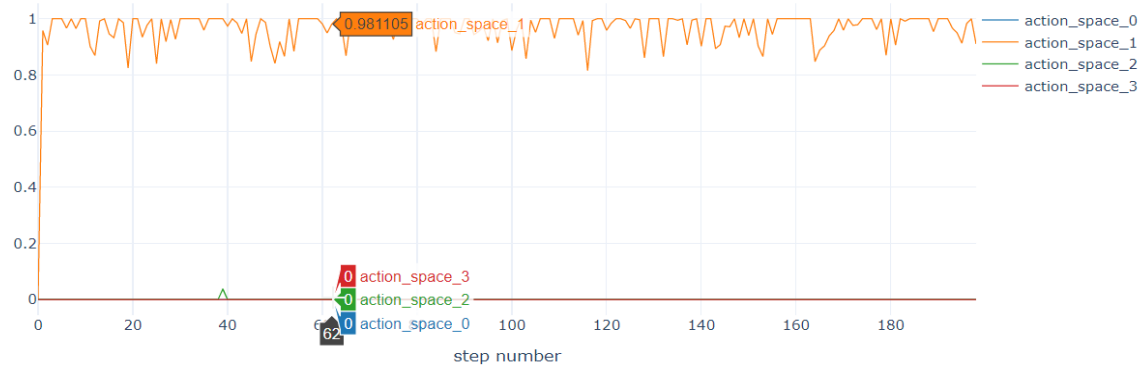
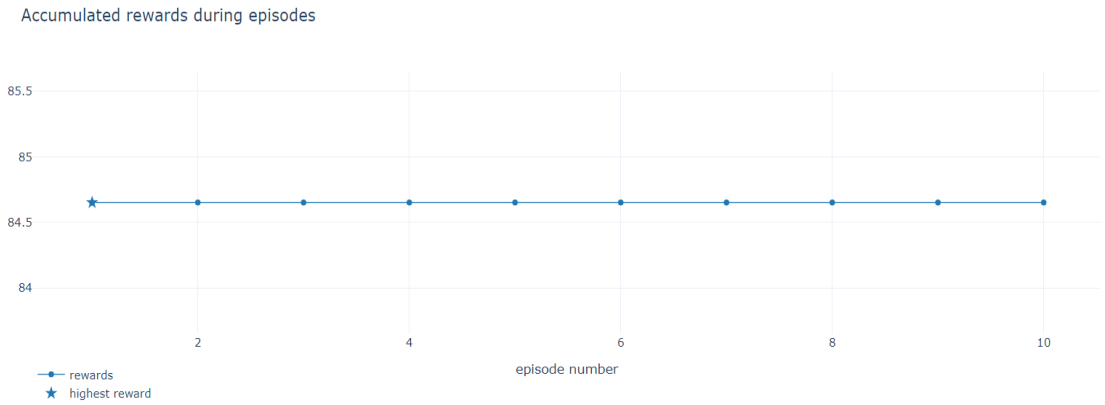


Figure 14. ***Training:*** *Best episode's action space.*

[7] Physically we can interpret a fractional value as the port of the valve being partially closed / open. So, for example a value of 0.8 as a valve almost fully open.

Testing

Regarding the testing the accumulated reward is slightly lower than the one of DQN, being 84.6.



Talking about the state space it's very similar to the one of DQN, but to see some differences we will help us with numerical measures as MSE and MAE.

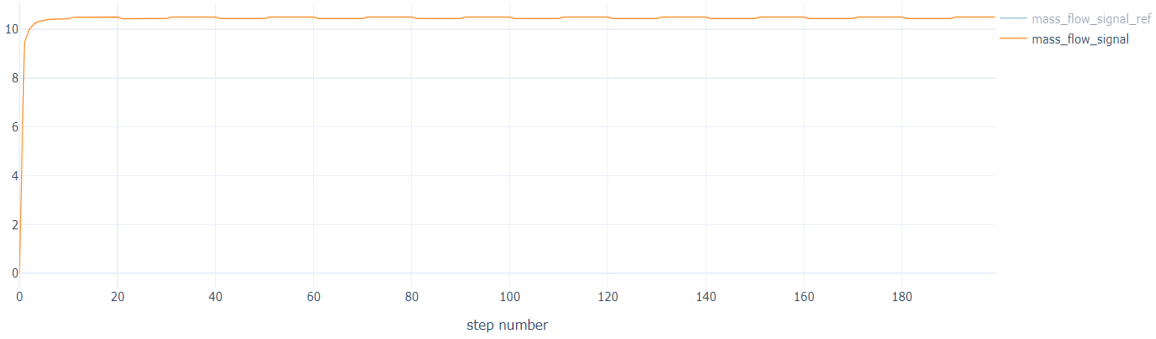


Figure 15. ***Testing:*** *state space.*

Interesting is the action space which, differently from the training, is showing us that the agent picked a value very close to 1 0.999 instead of the one before that was oscillating between 0.8 and 1, meaning that despite being able to choose fractional value of actions, the ones that yield to higher values are still close to 1.

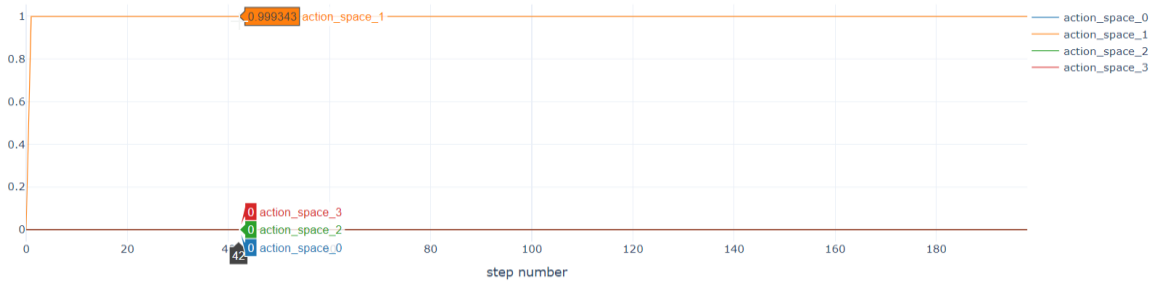


Figure 16. ***Testing:*** *action space.*

Regarding the error between the output and the reference we obtained these results.

MSE	MAE	Total error
0.56	0.095	19.09

For DDPG we used the default parameters. We have to notice that in order to control the balance between exploration and exploitation in DQN we tuned the exploration rate. Instead, to make DDPG policies explore better, we add noise to their actions at training time [11] using the initial noise and the decay noise rate.

Buffer size	1 000 000
Discount rate	0.9
Initial noise	0.1
Decay noise rate	0.0001
Critic Learning rate	0.001
Actor Learning rate	0.0005

Table 2. DDPG's parameters

Now we can finally discuss an agent for the complete system of 14 valves. We reduced the number of timesteps for each episode to 100 because otherwise the training / testing would have taken too much time.

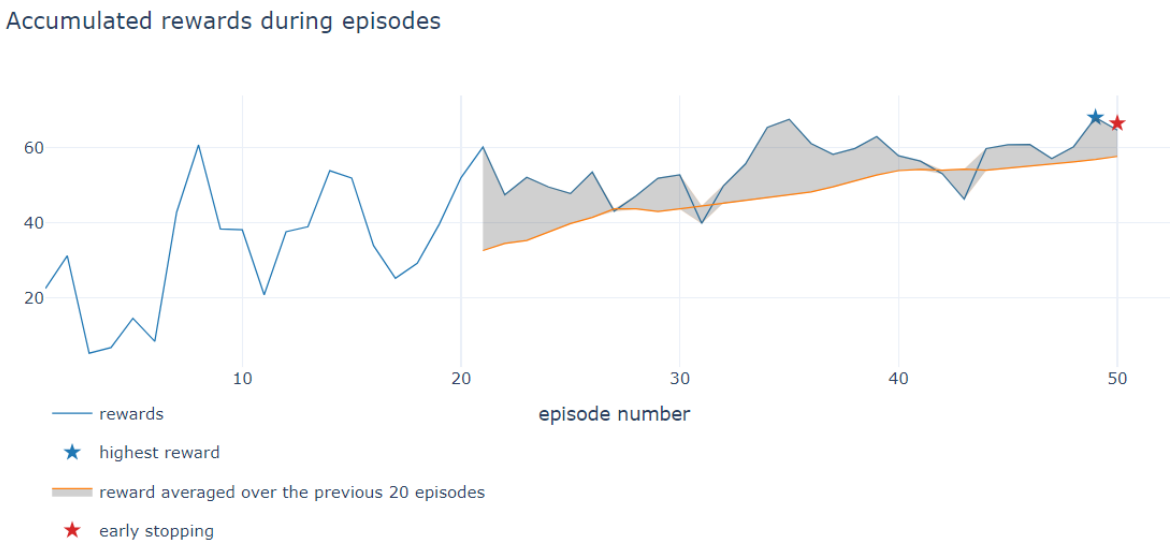


Figure 17.

Training of 100 episodes of 100 timesteps. Best episode: 49. Total reward of 66.

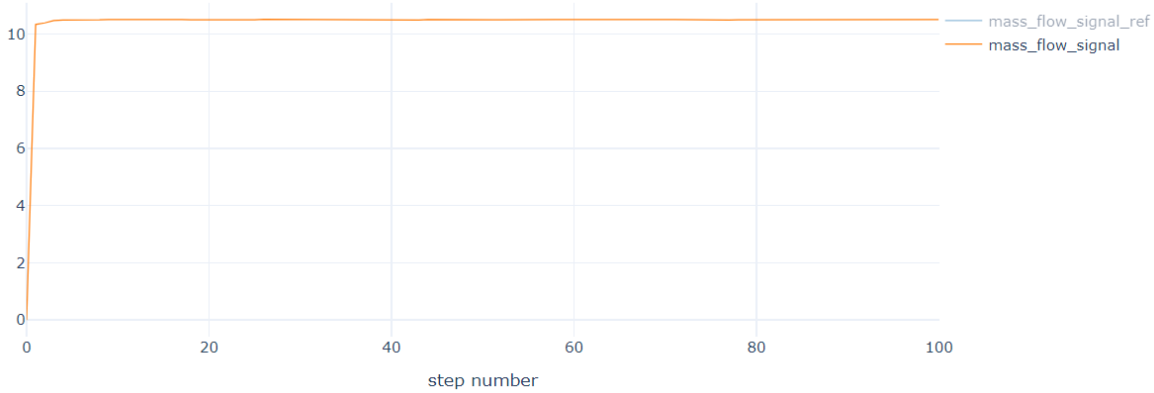


Figure 18. ***Training:*** *Best episode's state space.*

The action space presents 8 actions to 0 (8 valves fully closed) 2 actions to 1 (2 valves fully open), and the other 4 partially open. Still, we have the majority of valves fully closed.

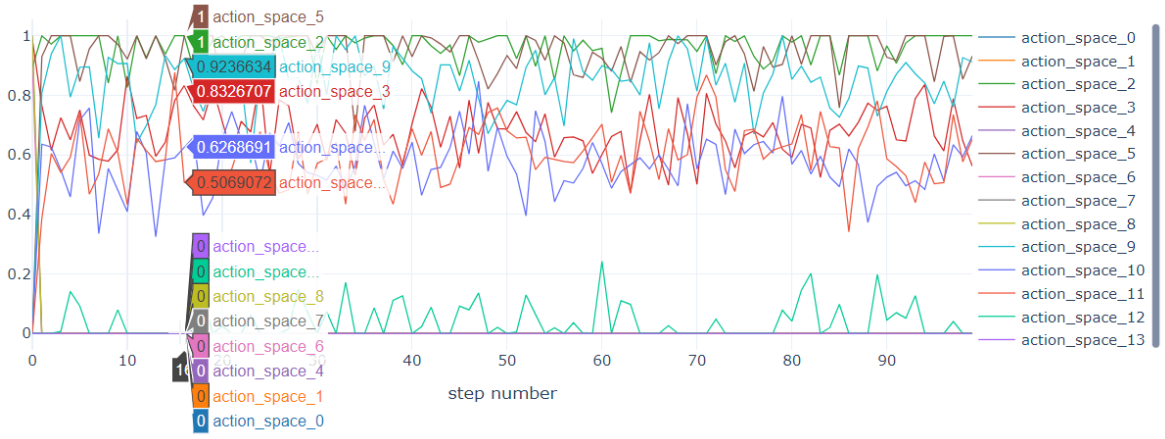


Figure 19. ***Training:*** *Best episode's action space.*

Testing

Evaluating the policy, we can see that the reward settles around 66.5.

The state space here seems almost perfect, in fact the two lines overlap. We can also see from Figure 21 that the signal is clearer, flatter than the previous ones.

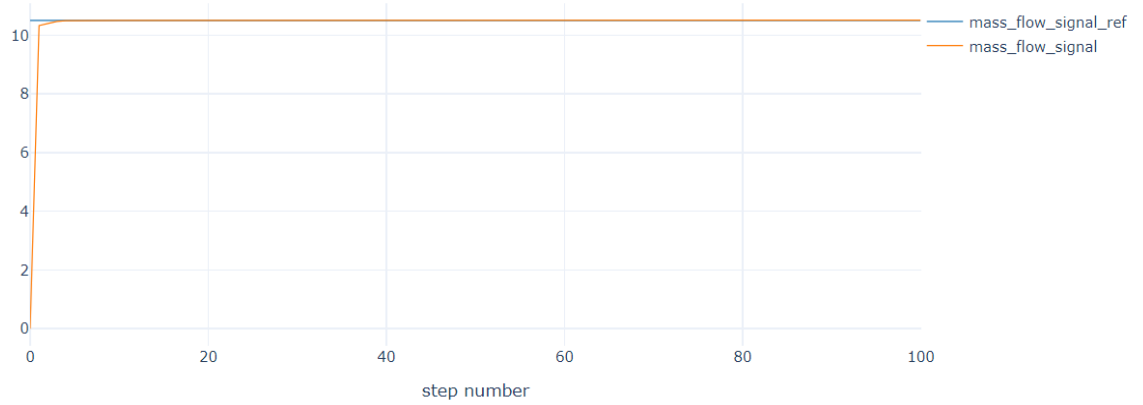


Figure 20. ***Testing:*** *state space.*

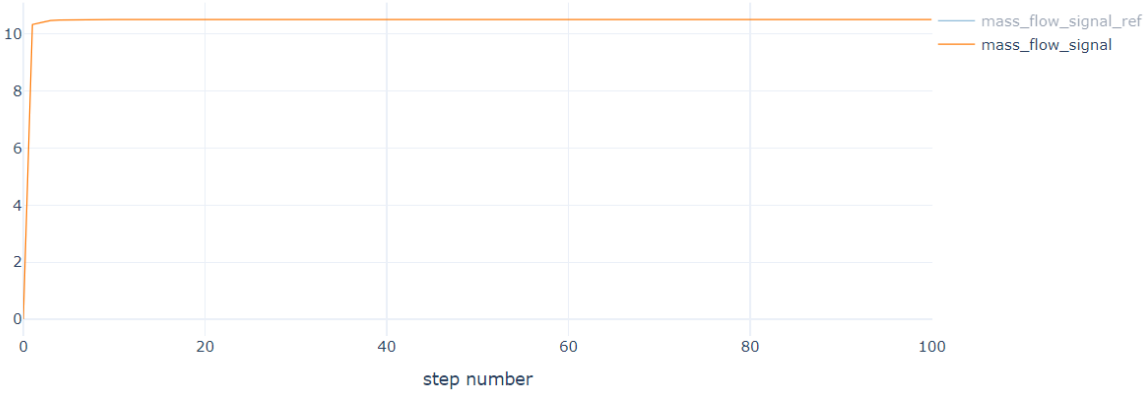


Figure 21. ***Testing:*** *state space.*

The action space reflects what we said earlier.

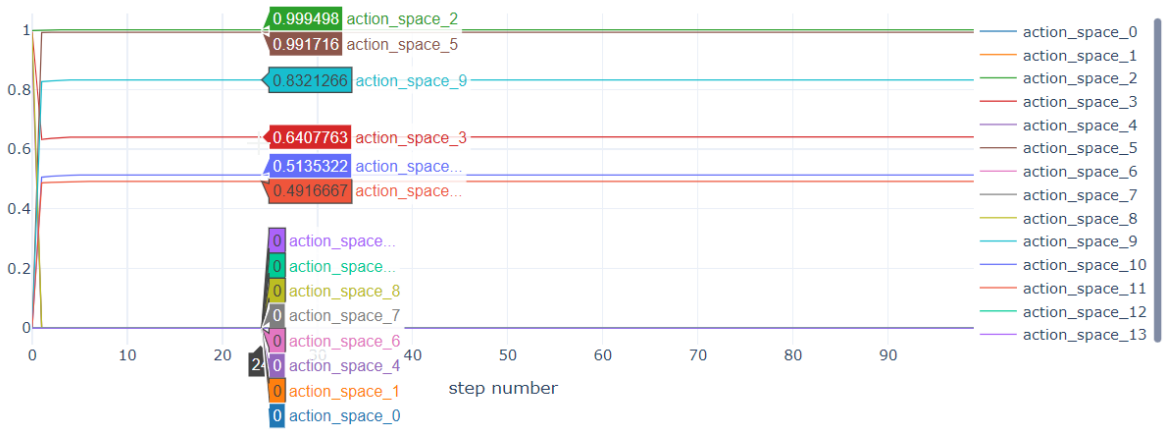


*Figure 22. **Testing**: action space.*

These are the results regarding the measurements of the error. We can't really say something since the system is different from the previous one.

MSE	MAE	Total error
1.09	0.11	11.27

Buffer size	1 000 000
Discount rate	0.9
Initial noise	0.1
Decay noise rate	0.0001
Critic Learning rate	0.001
Actor Learning rate	0.0005

Table 3. DDPG's parameters.

4.2 Second use case: Valve Line

At this point of our study, it was asked to us by the third-party company that commissioned the work, to focus our attention and what we learned until that point on another mechatronic system which we are going to call Valve Line.

Actually, this system and the Tubing Array are part of a bigger system.

The fundamental structure of the system is similar to the other one, since we are still talking about pneumatic valves.

Also, the objective of our algorithm is the same. We have to control the input of the valve in blue in order to have the output of the system, which this time is a pressure value, as close as possible to a reference that we define, so again a target following problem. However, there are two main differences: the first one is that the system has 3 valves / actuators (the ones circled in Figure 23) that change configuration during time, meaning that their inputs will variate and so the output will.

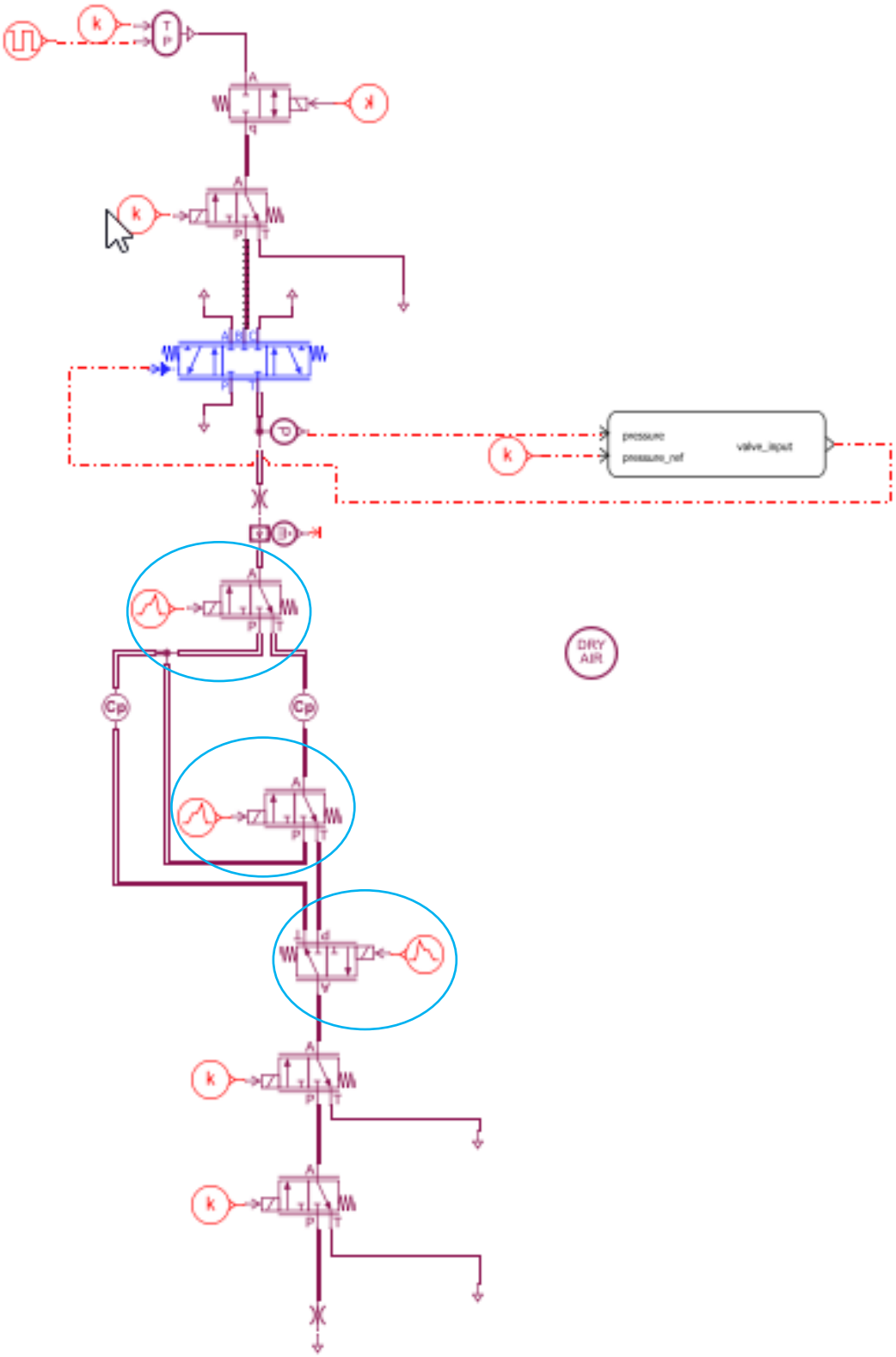


Figure 23. Valve Line system

The second difference is that now we have to control only one valve and the valve is different from the previous ones and also from the ones connected to it in the Valve Line system. In fact, we are talking about 5 way 3 positions.

The 3 positions are:

- **Blocked center;**
- **Open center;**
- **Pressure center.**

Blocked center

In this position, all the valve ports are blocked. Air cannot flow through the valve to either actuator port as the supply path to those ports is closed.

Air also cannot flow from either actuator port to either exhaust port as those flow paths are blocked. The supply, actuator, and exhaust ports are all closed. This position will correspond to action 1.

Open center

When the 5/3 air valve is shifted into its central position in an “Open Center” three position style valve, the supply line to the valve is blocked, and both cylinder ports are open through the valve to the exhaust. This position will correspond to action 0.

Pressure center

In the “Pressure Center” position, air will flow from the supply to both air actuator ports, and the exhaust port(s) are blocked. This position will correspond to action -1. [14]

4.2.1 DQN

Since we saw that DQN worked pretty well with the reduced system, and since is easier to tune due to the lower number of parameters (DDPG is an actor critic algorithm so we have the double number of parameters) and by the moment that we have only 1 valve to control, we will start with it. The possible actions that our agent can choose are the ones mentioned above: 1, 0 and -1.

Respectively to the first system we changed the reward function. In fact, the agent was not working properly with the one we previously defined. As we will see, due to the change of configuration the output of the pressure will variate a lot without following constantly the reference, so we wanted to find a way to penalize the agent when the pressure signal would fall down to lower values, very far from the reference.

We established a threshold of the error that was acceptable for us and decide to give a penalization (because the reward is negative) directly proportionate to how far the output went from the reference. On the other hand, if the gap was within the threshold, we gave a reward inversely proportionate to the error.

```
abs_error = abs(o_tp1[1] - o_tp1[0])
delta = 0.01/(abs_error+0.01)

inv_error = 1/abs_error+0.0001
penaliz = 0.5/(inv_error + 0.01)

if abs_error < 0.23:
    r_tp1 = delta
else:
    r_tp1 = -penaliz
```

Figure 24. Reward function for Valve Line system. The threshold is 0.23[8]

[8] The value of the threshold was found after some trainings we did. It could be set in a better way.

Training

The first thing that we can notice is the fact that the reward is negative. This because as we can see from the state space, even the one of the best episode in Figure 27, we have some ranges of timesteps 70-150 and 220-300 where the value of the output is very far from the reference, close to 0. So, in that part, in line with our reward function, we give a huge penalization, while in the other parts, where the gap is closer, the positive reward that we give is not enough to balance it.

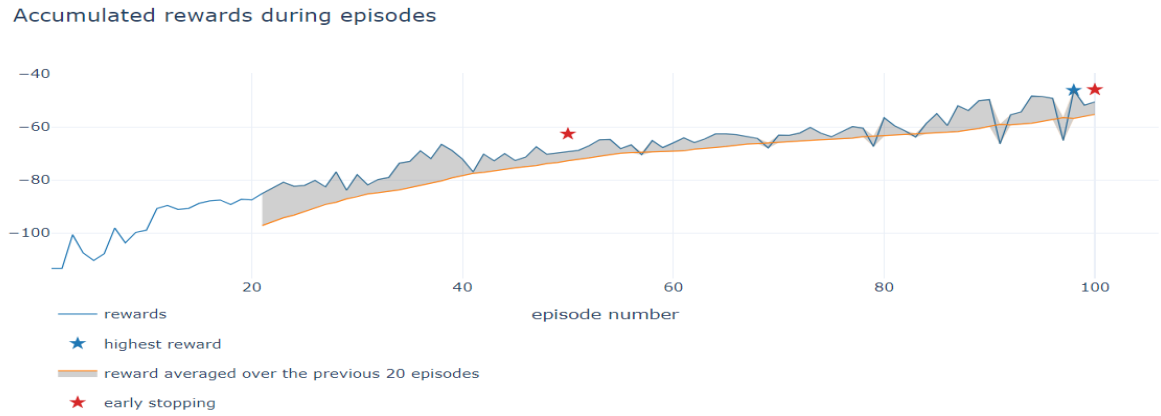


Figure 25.

Training of 100 episodes of 350 timesteps. Best episode 98. Reward of -46.

The trend although is quite good with the amount of total reward that is growing through the whole episode.

We can confront the state space of one of the first episodes and the one of the best episode.[9]

[9] Again, we have chosen the value of the reference in the same mode of the previous system.

As we can see in Figure 26 even when the value of the pressure (output) is higher than 0, we have a signal that is very unstable. What we wanted to do with the idea of the reward function was avoiding this behvior.

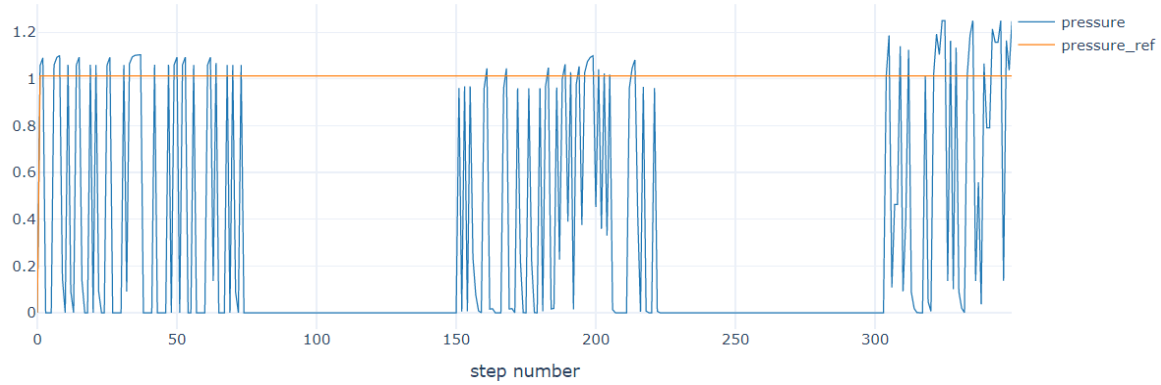


*Figure 26.**Training:** First episode's state space.*

By looking in Figure 27 we can see that we managed to obtain a constant signal in the 'block' where the pressure is close to the reference, also getting really closer in the last one, although we didn't manage to raise the signal in the other blocks.

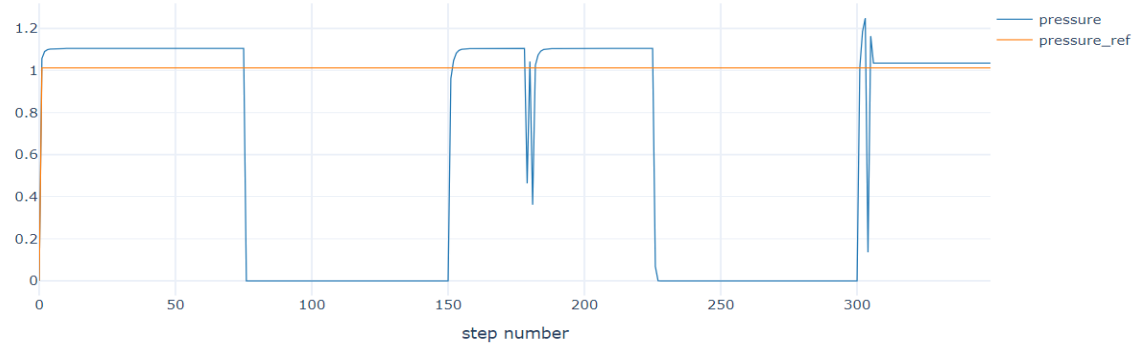


*Figure 27.**Training:** Best episode's state space.*

A similar behavior could be spotted by looking at the action space of the first episode which is very chaotic, which is in line with the trend of the state space showed in Figure 26. In fact, by picking actions quite randomly we have a bad output.

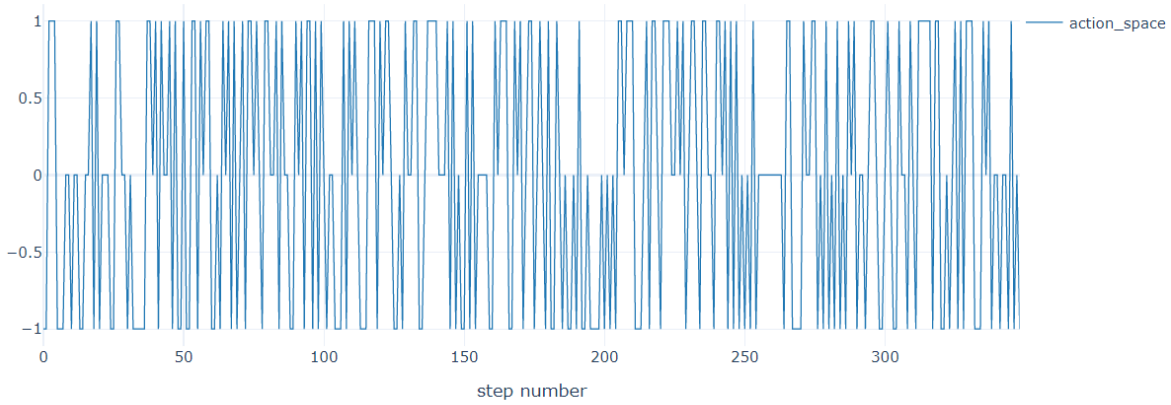


Figure 28. ***Training:*** *First episode's state space.*

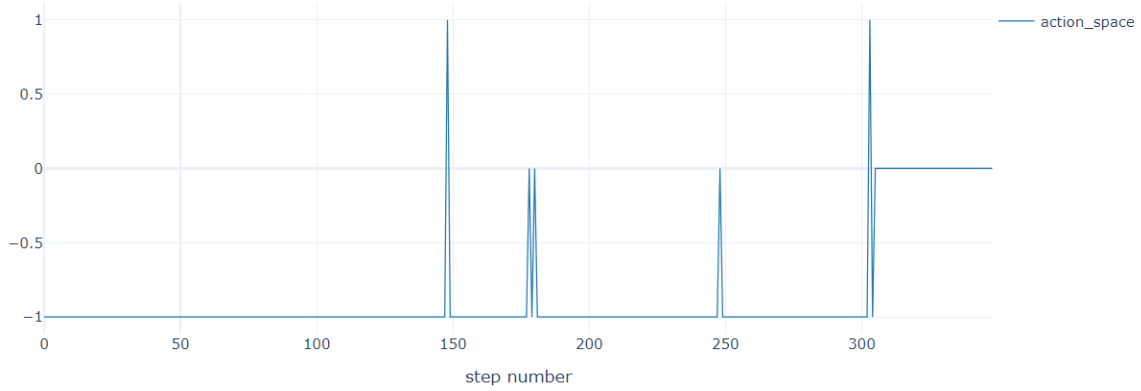


Figure 29. ***Training:*** *Best episode's state space.*

On the other hand, in best episode's action space, we can see that the agent learned that it could reach higher rewards by picking these actions.

Testing

The testing confirms what we saw, with also a better behavior in the last ‘block’.

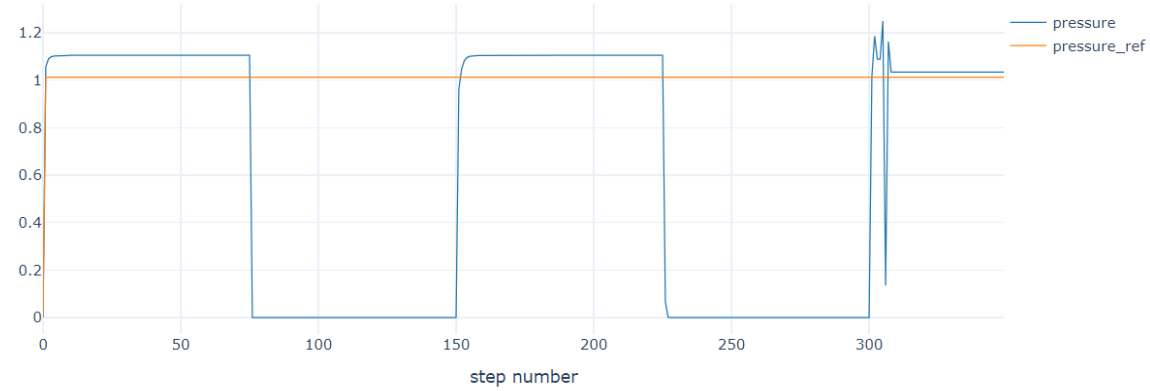


Figure 30. ***Testing:*** *state space.*

Also the action space is a bit different, in fact we have that the prevailing action is -1.

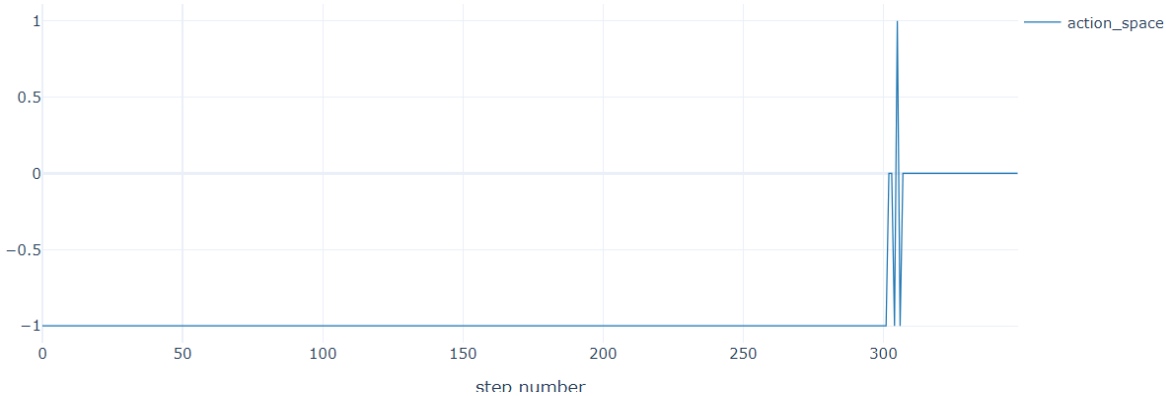


Figure 31. ***Testing****: action space.*

Results obtained with the algorithm:

MSE	MAE	Total error
0.44	0.48	167.9

Buffer size	1 000 000
Discount rate	0.9
Initial exploration rate	0.9
Decay exploration rate	0.05
Minimum exploration rate	0.1
Learning rate	0.0005

Chapter 5:

Conclusions

The aim of this study was to research the potential of the usage of Reinforcement Learning for mechatronic system control. As we said Reinforcement Learning is a recent yet growing technology in the area of machine learning, which already demonstrated a lot of potential in several fields, excelling in videogames and board games compared to humans.

However, as we said before, applying these technologies to the control of mechatronic system is challenging since we have to deal with model uncertainties, sometimes high nonlinearities, complicated coupling, stringent performance requirements and also because often one of the most important features for a mechatronic system is reliability.

However, the level of difficulty is proportional to the improvement that Reinforcement Learning could give to control these systems.

We opened our research with the first system given to us from the third-party agency that was called Tubing Array, a mechatronic system of pneumatic valves.

We started using DQN as the first algorithm to implement the Reinforcement Learning agent. I have to say that tuning the agent was pretty simple thanks to the amazing help Simcenter Studio gives us in developing Reinforcement Learning algorithm.

Firstly, we used an algorithm for a simplified system of 4 valves in order to better spot possible errors and to better tune the agent. As we saw in chapter 4 the DQN algorithm gave us pretty good results as our output managed to follow quite closely, even if with some gap, the reference that we defined. Talking about the reference, as we previously explained, we defined a value in line with the characteristics of the Tubing Array system, since the third-party company didn't give us any predefined value of the reference. We can sum up the results of this agent for this system:

Tubing Array 4 valves	
DQN	
Best reward	84.7
MSE	0.56
MAE	0.094
Total error	18.96

Table 4. DQN's results.

We know that both for MSE and MAE, and obviously for the total error, the closer we get to 0 the better our algorithm is, hence this confirms us that DQN algorithm for this system with 4 valves worked quite well, even if perhaps it would have been possible to get closer to 0, maybe finding a better set of hyperparameters, which as we said in chapter 2 is a laborious process that requires many iterations and is computationally expensive to find the best settings that produce the best neural network architectures.

As we wanted to focus our attention on the system of 14 valves, the actual system, we encountered some problems with the usage of DQN related to the high number of actions that the agent should have evaluated. In fact, being its action space discrete and having the

system 2 inputs for each valve and a total number of 14 valves it should have choose among 2^{14} = 16384 possibilities.

Considering that, we decided to proceed with another algorithm DDPG, which among its features, has as the most relevant ones the fact of being an Actor-Critic algorithm which operates with a continuous action space, thing that allowed us to tune an algorithm for the system of 14 valves.

Despite that, we still decided to firstly begin with the reduced system of 4 valves, for the same reasons of DQN. Again, the results of DDPG are summed up in the table:

Tubing Array 4 valves	
DDPG	
Best reward	84,6
MSE	0.56
MAE	0.095
Total error	19.09

Table 5. DDPG's results.

We can confront the results of DQN and DDPG for the 4 valves system:

Tubing Array 4 valves		
	DQN	DDPG
Best reward	**84.7**	84.6
MSE	0.56	0.56
MAE	**0.094**	0.095
Total error	**18.96**	19.09

Table 6. 4 valves system algorithm comparison.

As we can notice the results of DQN are slightly better than the ones of DDPG, expect for the MSE that is the same. In fact, as the goal in Reinforcement learning is to maximize the reward, the higher it gets the better is. Moreover, regarding MSE, MAE and Total error, which are all measurements of the error, the lower they are the better is.

In conclusion, regarding the 4 valves system DQN gave better results and was also easier to tune and required less computational time.[10]

Shifting the attention to the system with 14 valves, we finally could produce some results, which, regarding the state space seemed to be pretty good at sight, even if calculating the MSE and the other ones it seemed to be worse than the ones we got with the system with 4 valves, but that could be due to the increased complexity of the system.

Tubing Array, 14 valves	
DDPG	
Best reward	66
MSE	1,09
MAE	0,11
Total error[11]	11,27

Table 7. 14 valves, DDPG's results.

[10] It means the time needed to end the training / testing.
[11] The value of the total error is lower than the one of the previous algorithms just because it refers to an episode with 100 timesteps, half of the previous ones.

We could have done more and could have tried to develop better algorithms for our systems; however, the third-party company asked us to focus on the new system, called Valve Line, described in chapter 4.2.

We didn't have a lot of time to dedicate to this new system thus, the algorithms whit which we worked could have been tuned better.

In fact, one of the most important features that we changed respect to the previous algorithms is the reward function and, as we said in chapter 2, reward shaping is one of the biggest challenges when talking about Reinforcement Learning.

Both from the state space and from Table 8 below we can see that DQN was quite good even if we have a huge total error due to the part where the signal falls to 0.

Valve Line	
	DQN
Best reward	**-46**
MSE	**0.44**
MAE	**0.48**
Total error	**167.9**

Table 8. Valve Line results

Despite all, for matter of time we weren't able to understand if the 'losses' of signal manifested in the state space (when the pressure signal goes down to 0), both in Figure 30 could have been corrected / 'raised' with a better configuration of the algorithm. We have to consider that the behavior could be due to the structure of the system itself. In fact, we have to remember that in this system the valves configuration changes in time, producing variations in the output pressure. The question that we couldn't answer is: can we correct with the algorithm that behavior?

In conclusion we produced interesting results to approach the control of mechatronic systems with Reinforcement Learning, using some of the basilar Reinforcement Learning algorithm.

Probably by having more time and a deeper knowledge of the sector we could have produced results more solid and reliable.

We can say that in the future there will be a lot of space for Reinforcement Learning as this field is in constant development, which will also include the control of mechatronic systems.

In conclusion, it can be said that Reinforcement Learning is one of the most interesting technologies nowadays, because it approaches learning in the most similar way to how a human being would, who learns by trial and error, in the absence of prior knowledge.

Chapter 6:

Abstract in lingua italiana

L'obiettivo di questo studio si concentra nel ricercare algoritmi di Reinforcement Learning per il controllo di sistemi meccatronici. Il Reinforcement Learning è un tipo di tecnica di apprendimento automatico in cui un agente informatico impara a eseguire un'attività attraverso ripetute interazioni fatte da tentativi ed errori con un ambiente. Questo approccio di apprendimento consente all'agente di prendere una serie di decisioni con l'obiettivo di massimizzare una ricompensa per il compito senza l'intervento umano e senza essere esplicitamente programmato per raggiungere il compito.

Il Reinforcement Learning può essere applicato ad una grande varietà di campi di ricerca: dai videogames e giochi da tavolo, ai robot, alla guida autonoma, problemi di schedulazione, etc.

Al momento in cui si scrive questo lavoro, il Reinforcement Learning è stato oggetto di numerosi studi, come quello condotto da Deep Mind nel 2015, nel quale è stato presentato il primo agente artificiale nel campo del Reinforcement Learning in grado di raggiungere performance pari a quelle umane in videogiochi come ATARI, oppure, come dimostrato in un altro articolo dei ricercatori di Deepmind in giochi da tavolo come scacchi.

Il nostro studio riguardava la possibilità di controllare un sistema meccatronico tramite un algoritmo di Reinforcement Learning.

Il primo sistema, chiamato (Tubing Array) fornitoci da un'azienda esterna era composto da 14 valvole pneumatiche connesse tra loro.

Il sistema aveva come input due valori costanti di pressione e temperatura e produceva un segnale di flusso di massa. Il nostro obiettivo era quello di controllare gli input di ciascuna delle valvole tramite un algoritmo di Reinforcement Learning in modo tale che il segnale di output prodotto dal sistema fosse il più vicino possibile ad un valore costante di riferimento definito da noi.

Per fare ciò abbiamo usato due software: Amesim e Simcenter Studio. Il primo per visualizzare e lavorare sulla struttura del sistema e per creare un'interfaccia che ci permettesse poi di collegare gli input delle varie valvole con il nostro algoritmo, il quale invece è stato sviluppato grazie a Simcenter Studio.

Il primo algoritmo che abbiamo scelto è il DQN, il quale si basa sul Q-learning, una politica di Reinforcement Learning che troverà la prossima azione migliore, dato lo stato attuale. Sceglie questa azione a caso e mira a massimizzare la ricompensa. Il DQN affianca al Q-learning l'uso di reti neurali, le quali ci permettono di applicare i nostri algoritmi ad ambienti con molte variabili e molte azioni da valutare.

Per vedere se il nostro algoritmo funzionasse senza errori e per facilitarne la sua configurazione abbiamo preferito testare il nostro algoritmo dapprima con un sistema di 4 valvole anziché 14.

Con questo sistema ridotto il DQN ha prodotto buoni risultati, i quali possono essere evidenziati dai grafici che descrivono lo spazio di stato durante l'episodio prodotto dal test del nostro algoritmo, in Figura 10 e da quello che descrive lo spazio delle azioni durante

l'episodio prodotto dal test, in Figura 11 del capitolo 4. Infatti, l'output del sistema segue il valore costante di riferimento discostandosi da esso di poco.

Per poter meglio valutare l'errore, il divario fra questi due valori ci siamo serviti di alcune misurazioni dell'errore come l'MSE il MAE e l'errore totale (la differenza tra l'output e il valore di riferimento) durante l'episodio.

Da questi valori, sotto riportati, possiamo vedere che l'algoritmo abbia avuto un buon comportamento, visto che il valore ideale sia dell'MSE che del MAE è 0.

MSE	MAE	Total error
0.56	0.094	18.96

Nonostante ciò, abbiamo dovuto usare un altro algoritmo per il sistema completo con 14 valvole. Questo perché avendo 14 valvole e 2 input possibili per ognuna (o valvola chiusa, 1 aperta), il nostro algoritmo DQN avrebbe dovuto valutare ad ogni step circa 16000 azioni, il che avrebbe comportato problemi di natura computazionale e di memoria.

Perciò abbiamo scelto un altro algoritmo, il DDPG, il quale oltre ad operare con uno spazio d'azioni continuo (non discreto come il DQN), è un algoritmo di tipo Actor-Critic. Infatti, la struttura della politica è nota come attore (Actor), perché è utilizzata per selezionare le azioni, e la funzione del valore stimato è nota come critica (Critic), perché critica le azioni compiute dall'attore.

Anche questo algoritmo ha mostrato risultati buoni allo stesso modo del DQN anche se quest'ultimo si è rivelato leggermente migliore, come possiamo notare dalla seguente tabella:

Tubing Array 4 valves		
	DQN	DDPG
Best reward	**84.7**	84.6
MSE	0.56	0.56
MAE	**0.094**	0.095
Total error	**18.96**	19.09

A questo punto abbiamo deciso di valutare l'algoritmo DDPG con il sistema completo composto da 14 valvole. Anche in questo caso abbiamo avuto un comportamento similare ai due precedenti, notando particolarmente che, seppur potendo l'algoritmo scegliere azioni continue tra 0 e 1, decide di mantenere 8 valvole chiuse (input a 0) e le altre a valori vicini ad 1, come possiamo vedere in Figura 22 del capitolo 4.

Nonostante dallo spazio di stato in Figura 21 l'errore sembri molto piccolo, i risultati seguenti evidenziano come esso in realtà sia più alto, anche se dobbiamo tenere in considerazione che non possiamo realmente confrontare gli algoritmi in quanto fanno riferimento ad un sistema diverso.

Tubing Array, 14 valves	
	DDPG
Best reward	66
MSE	1,09
MAE	0,11
Total error[12]	11,27

[12] The value of the total error is lower than the one of the previous algorithms just because it refers to an episode with 100 timesteps, half of the previous ones.

A questo punto l'azienda esterna ci ha comunicato di spostare la nostra attenzione su un altro sistema, chiamato Valve Line. Questo sistema riguardava ancora un sistema di valvole pneumatiche, ma con l'obiettivo di controllare una singola valvola, in modo da produrre un output (una pressione) il più vicino possibile ad un riferimento costante.

Questa valvola da controllare aveva delle caratteristiche differenti. Infatti, si tratta di un tipo di valvole capaci di assumere tre differenti posizionamenti, i quali abbiamo deciso di rappresentare con 3 azioni: -1, 0, 1.

Una delle differenze maggiori che possiamo notare in questo sistema riguarda il segnale di pressione in uscita. Infatti, il segnale cade più volte vicino allo 0, per poi tornare a valori vicini a quello di riferimento. Nonostante ciò, in alcuni punti, soprattutto l'ultimo (in Figura 30), l'algoritmo il segnale si riavvicina al riferimento.

Nei risultati prodotti dall'algoritmo mostrati in tabella possiamo notare che il valore della ricompensa più alta è negativo. Questo è dovuto al fatto che per l'algoritmo di questo secondo sistema, abbiamo utilizzato una funzione per la ricompensa, la quale penalizzava l'agente ogni qual volta esso produceva un output che si allontanasse troppo dal riferimento.

Inoltre, il valore della penalità era direttamente proporzionale al valore dell'errore tra l'output e il riferimento, perciò visto che in certi punti questo era molto ampio, abbiamo ottenuto una ricompensa negativa.

Valve Line	
	DQN
Best reward	**-46**
MSE	**0.44**
MAE	**0.48**
Total error	**167.9**

Nonostante tutto, per questioni di tempo non siamo riusciti a capire se le 'perdite' di segnale manifestate nello spazio degli stati (quando il segnale di pressione scende a 0 – cfr. Figura 30) avrebbero potuto essere corrette con una migliore configurazione dell'algoritmo. Bisogna considerare che il comportamento potrebbe essere dovuto alla struttura del sistema stesso. Ricordiamo infatti che in questo sistema la configurazione delle valvole cambia nel tempo, producendo variazioni della pressione di uscita. La domanda a cui non siamo riusciti a rispondere è: possiamo correggere con l'algoritmo questo comportamento?

In conclusione, abbiamo prodotto risultati apprezzabili per il controllo di sistemi meccatronici attraverso Reinforcement Learning, utilizzando algoritmi piuttosto semplici.

Probabilmente avendo a disposizione un tempo maggiore e una più approfondita conoscenza del settore si potrebbero avere risultati solidi e affidabili.

Possiamo affermare che nel futuro ci sarà molto spazio per il Reinforcement Learning essendo questo campo in costante sviluppo, il quale includerà anche il controllo dei sistemi meccatronici.

In conclusione, si può affermare che il Reinforcement Learning è una delle tecnologie più interessanti al giorno d'oggi perché si avvicina all'apprendimento nel modo più simile a come farebbe un essere umano, che apprende per tentativi ed errori, in assenza di una conoscenza pregressa.

Bibliography

[1] Barto. A. G., Sutton R. S., Reinforcement learning: An introduction, MIT press, 2018.

[2] Syuan-Yi C., «Applications of Intelligent Control Methods in Mechatronic Systems.,» *MDPI Journal,* 2020.

[3] Houda B., Fenjiro Y., «Deep Reinforcement Learning Overview of the state of the Art,» *Journal of Automation, Mobile Robotics and Intelligent Systems,* 2018.

[4] Kavukcuoglu K., Silver D., Graves A., Antonoglou I., Wierstra D., Riedmiller M. Mnih V., «Playing Atari with Deep Reinforcement Learning,» 2013.

[5] Schrittwieser J., Simonyan K., Antonoglou I., Huang A., Guez A., Hubert T., Baker L., Lai M., Bolton A., et al. Silver D., «Mastering the game of go without human knowledge,» *Nature,* 2017.

[6] Lefebvre T., Crevecoeur G., Staessens T., «Adaptive control of a mechatronic system using constrained residual reinforcement learning,» 2021.

[7] Yuxi L., «Reinforcement Learning in Practice: opportunities and challenges,» 2022.

[8] «Part 1: Key Concepts in RL,» 2018. [Online]. Available: https://spinningup.openai.com/en/latest/spinningup/rl_intro.html..

[9] «Part 2: Kinds of RL Algorithms,» 2018. [Online]. Available: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html#part-2-kinds-of-rl-algorithms.

[10] «Deep Learning,» 2020. [Online]. Available: https://www.ibm.com/cloud/learn/deep-learning.

[11] Zhou V., «Machine Learning for Beginners: An Introduction to Neural Networks,» 2019. [Online]. Available: https://victorzhou.com/blog/intro-to-neural-networks/. .

[12] «Deep Deterministic Policy Gradient,» 2018. [Online]. Available: https://spinningup.openai.com/en/latest/algorithms/ddpg.html.

[13] Seif G., «Understanding the 3 most common loss functions for Machine Learning Regression,» 2019. [Online]. Available: https://towardsdatascience.com/understanding-the-3-most-common-loss-functions-for-machine-learning-regression-23e0ef3e14d3.

[14] Aidan W., «5 3 Valves Explained – 5 Way 3 Position Pneumatic Solenoid Valves GUIDE,» [Online]. Available: https://www.about-air-compressors.com/5-3/#:~:text=A%205%203%20valve%20is%20a%20directional-control%20valve,to%20end.%20What%20is%20a%205%20port%20valve%3F.