UNIVERSITÀ POLITECNICA DELLE MARCHE

Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria Meccanica

REINFORCEMENT LEARNING FOR CONTROL OF MECHATRONIC SYSTEMS

REINFORCEMENT LEARNING PER CONTROLLO DI SISTEMI MECCATRONICI

Relatore:
Prof. Nicola PAONE

Candidato:
Federico CARNEVALINI

Correlatore:
Dr. Jonathan MENU

Anno Accademico 2020/2021

ABSTRACT (IT)

ABSTRACT (IT)

- La maggior parte dei sistemi elettromeccanici con cui gli esseri umani interagiscono ogni giorno necessitano di modificare il loro comportamento in funzione di determinati parametri di input. Che si tratti di controllare la temperatura di una stanza con un semplice termostato o regolare la coppia motrice di un motore, è sempre necessaria la messa a punto di una funzione di trasferimento che determini il corretto output da inviare agli attuatori per ottenere l'effetto desiderato. Progettare un sistema di controllo automatico che, come nel caso degli esempi precedenti, consenta l'attuazione di una certa operazione in uscita in funzione di una o due variabili in ingresso è oggi pratica estremamente comune; i classici sistemi in retroazione PID, ad esempio, consentono nella maggior parte dei casi di raggiungere questi scopi, ma con l'avvento della robotica e la diffusione di sistemi sempre più complessi questi potrebbero non essere più sufficienti.

Affinché robot possano operare in ambienti mutevoli dovrebbero essere in grado di variare dinamicamente il loro comportamento al fine di raggiungere l'obiettivo prefissato. In altre parole, la funzione di trasferimento del controllore dovrebbe variare continuamente al variare delle condizioni ambientali. Questa necessità, unita all'aumento di potenza computazionale dei computer nell'ultimo decennio, ha portato allo sviluppo di tecnologie innovative che rappresentano un completo cambio di paradigma rispetto alle tecniche più classiche: anziché programmare controllori per eseguire un certo output in funzione di certi input si passa allo sviluppare sistemi in grado di auto-programmarsi e trovare il miglior modo di ottenere degli obiettivi, indipendentemente dalle condizioni ambientali.

- Nel mondo dell'intelligenza artificiale, il Reinforcement Learning è una tecnica di Machine Learning che ha acquistato enorme popolarità negli ultimi anni, sia per utilizzi legati alla ricerca accademica ed all'analisi dei dati, che per altre applicazioni più pratiche in vari settori quali ad esempio ingegneria e finanza. Questa tecnica punta alla realizzazione di agenti autonomi in grado di scegliere le azioni da compiere per il raggiungimento di determinati obiettivi tramite interazione con l'ambiente in cui sono immersi.

- L'obiettivo di questa tesi, svolta in collaborazione con Siemens Digital Industry Software, è quello di investigare l'applicabilità del Reinforcement Learning per il controllo di sistemi complessi e tentare di comprendere come la scelta di diversi algoritmi e parametri ne influenzino le prestazioni. Essendo un'applicazione relativamente innovativa di questa tecnologia non vi è a priori alcuna certezza di funzionamento, né un chiaro metodo di prosecuzione verso l'ottimizzazione degli algoritmi.

- Nella prima parte dell'elaborato sono presentati in maniera non troppo approfondita gli elementi teorici necessari a comprendere i principi di funzionamento del Reinforcement Learning e le differenze tra i tre algoritmi maggiormente utilizzati in questo lavoro. A seguire vengono illustrati gli strumenti principali utilizzati durante le attività, in particolare il software di sviluppo di sistemi meccatronici Simcenter Amesim, utilizzato per modellare i sistemi fisici da controllare, ed il software di progettazione generativa Simcenter Studio, utilizzato per scrivere ed eseguire codice in linguaggio Python che, con l'aiuto di apposite librerie dedicate al Reinforcement Learning, consente di simulare in interoperabilità con Amesim gli effetti dell'attuazione di qualunque algoritmo di controllo su un modello fisico 1D.

Successivamente si presentano gli esiti di alcune simulazioni, in particolare:

- un primo studio eseguito sul modello di un pendolo inverso su carrello, in cui si comparano le prestazioni del classico controllo in retroazione PID ad un algoritmo Q-Learning (algoritmo basato sulla memorizzazione tabulare di un valore "Q", rappresentante il reward totale atteso per ogni combinazione di stato del sistema ed azione eseguibile, determinato scegliendo la miglior sequenza possibile di azioni a partire da un determinato stato e calcolato attraverso un addestramento in cui il sistema misura e somma i reward provando le varie combinazioni di azioni per ogni stato);
- un secondo studio eseguito sul modello di un ascensore per uso civile, in cui si cerca di comparare l'efficacia di due altri algoritmi di Reinforcement Learning che sono il Deep Q-Learning (o DQN, di funzionamento simile al

Q-Learning ma che utilizza una rete neurale anziché una tabella per approssimare i valori "Q" attesi da ogni possibile azione del sistema per ogni possibile stato) ed il DDPG (simile al precedente teoricamente più adatto a sistemi con spazi di azione continui in quanto per ogni stato iniziale del sistema viene scelta l'azione più opportuna non da un set di azioni precedentemente definito bensì direttamente dallo stato corrente, con un criterio probabilistico).

In pratica, dopo aver accuratamente preparato il codice Python per eseguire le simulazioni desiderate, si vuole analizzare la capacità del sistema di controllo a seguire i setpoint al variare dell'algoritmo scelto ed al variare dei parametri che quest'ultimo utilizza durante il processo di apprendimento.

Si proveranno quindi svariate combinazioni, cercando di dedurre con un approccio sperimentale quali siano gli effetti dei vari parametri sulla bontà del controllo e anche provare a definire quale possa essere il miglior compromesso per raggiungere gli obiettivi impostati (es. inseguimento di traiettoria).

TABLE OF CONTENTS

1. BACKGROUND

1.1 MACHINE LEARNING

- In the setting of Artificial Intelligence, Machine Learning is the subject that studies how to develop computer algorithms that give to computers the ability to learn and automatically improve their ability to accomplish specific tasks by means of experience and without being explicitly programmed to do so. The data used by the computer to learn can be provided to the algorithm, or directly observed by it.

The greater the amount of data available, and the longer the training period, the more accurate the performed tasks will be. However, since it is impossible to provide an infinite amount of data and train for an infinite time, a core objective of a good learner is the ability to generalize from its experience. In other words, the algorithm must be able to perform accurately also on unseen situations, deducing how to behave from what it previously learned during the training.

From that observation, it appears obvious how the good performances of a learning machine are strictly correlated with the provision of a variegated and complete amount of different training situations.

- Depending on the nature of the input signal available to the learning system, Machine Learning can be divided in three main categories:

- *Supervised Learning*: the computer receive data containing inputs and desired outputs. The goal is to learn a general rule that maps inputs to outputs.
- *Unsupervised Learning*: the computer receives a set of data containing only inputs. It is his job to find some structure in the data, like grouping or clustering of data points.
- *Reinforcement Learning*: a computer program interacts with a dynamic environment in which it must reach a certain goal. As it navigates through space, the computer receives feedback in the form of observation of a new state representing the outcome of the taken actions. The new state is then converted into a numerical reward which is what it tries to maximize in the future actions.

1.2 ARTIFICIAL NEURAL NETWORKS

- In many real-world cases, Machine Learning algorithms need to be applied to situations where large state spaces are involved. When this happens, the computational time needed to compute all the data can become very high, as well as the memory needed to store all the values calculated from the functions in a tabular way. It is therefore necessary to improve that method and find a way to manage large amount of data even using limited computational resources. This can be done approximating these functions to give them the ability to approach new situations by generalizing from the experience they made on previous encounters with different states but in some sense similar to the new one.

- *Artificial Neural Networks* are widely used as nonlinear function approximators. They are a network of interconnected units like human neurons and are indeed modelled to resemble the behaviour of the human brain. The neurons are arranged in layers where each neuron takes some inputs and gives out an output to its connections. Generally, the network consists of an input layer, an output layer, and one or more hidden layers.

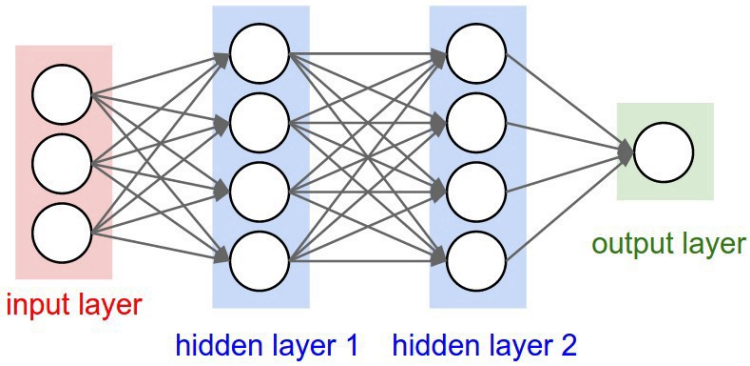


Fig. 1: A feedforward Artificial Neural Network

- The picture shows an example of a neural net where output signal from a unit never influences its input. This configuration is called *Feedforward Neural Network*. Each input signal is multiplied by variable weight. The output is the weighted sum of the input signals.

- To help the neural network in learning non-linear representations from the training data, an *Activation Function* (Sigmoid, Hyperbolic Tangent, ReLu) is introduced before the output of each neuron. This allows having a non-linear or semi-linear behaviour from the neurons. The reason why Activation Functions are required is that without them the behaviour of the series of neurons would be linear, and for the properties of linear mathematical transformations the entire network, no matter how deep, would be equivalent to a network with only two layers.

- Training an artificial neural network means changing the weights of each connection to improve the capacity of approximating the function that it is trying to learn. The idea is to start with a random initialization of the weights and calculate the output for a given input. The error between the generated output and the actual output is used to update the weights of the network.

1.3 REINFORCEMENT LEARNING

1.3.1 FUNDAMENTALS

- *Reinforcement Learning* is a branch of Machine Learning which deals with sequential decision making. It consists of an *agent* and an *environment*. The agent is the learner and the decision maker that acts on the environment, which responds to the action presenting a new situation to the agent and a scalar reward, according to a function defined by the operator. For each given situation, the agent will explore and try to optimize its actions in order to maximize the reward, in a continuous interaction between the two parts.

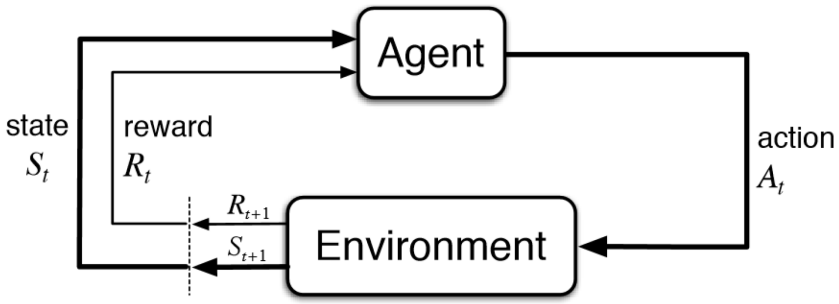


Fig. 2: Agent – Environment interaction

- An agent is considered good when it tries to maximize not only the immediate reward from the following action, but the global reward obtained once the objective of the system is reached.

1.3.2 MARKOV DECISION PROCESS

- Markov Decision Processes are a classical formalization of a sequential decision making, where a set of *states* $s \in S$ $(S = state\ space)$ is linked to a set of *actions* $a \in A$ $(A = action\ space)$ through a reward r and a transition probability:

$$p(s' \mid s, a) = \Pr\left(s_{t+1} = s' \mid s_t = s, a_t = a\right)$$

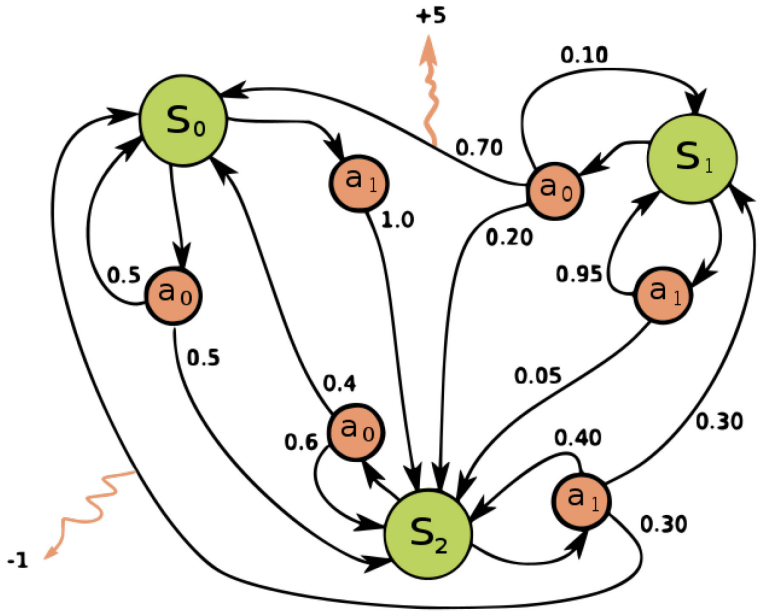


Fig. 3: Example of a MDP with three states (green circles) and two actions (orange circles), with two rewards (orange arrows)

- The goal of a Markov Decision Process is to find a good *policy* for the decision maker, so a correct distribution of probabilities between spaces and actions that will maximize, at the end of the process, the sum of the obtained rewards. The number of episodes required to finish this optimization process is not fixed, and it depends on the chosen algorithm, the environment complexity, and the presence or not of *early-stoppings* (threshold values that interrupt an episode if the results are diverging or interrupt the simulation if the reached policy is good enough).

1.3.3 POLICY

- The behaviour of the agent depends on the *policy*. As seen, the policy is the map determining what action to take at each state. It can be *deterministic*, when the execution of an action is guaranteed and depends only on the state:

$$\pi(s_t) = a_t$$

Or it can be *stochastic*, when the policy is nothing but a probability of success of some action for a given state:

$$\pi(a_t|s_t) = p_i$$

- In theory, a policy gathering the most amount of rewards possible from a given starting point is defined as *optimal policy* and is denoted by π^*.

1.3.4 REWARDS

- The peculiarity of Reinforcement Learning is the fact that the agent knows only what it needs to achieve, but not how to achieve it. It is indeed his task to figure out the best way to reach the result and learn a policy that allows it to act to do so in every situation.

To do so, a *reward* r_t in form of a numerical score is introduced: assigning positive rewards to desirable behaviours and negative rewards to the undesirable ones will lead the agent towards building a policy that maximizes the expected *accumulate sum of rewards* R_t for each different *timestep* t

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

- Sometimes, however, a future reward may have a different present value. In this case, a *discount factor* $0 \leq \gamma \leq 1$ is introduced. This is a learning parameter that basically just increases ($\gamma \to 0$) or decreases ($\gamma \to 1$) the importance of present actions over the future ones. Setting $\gamma = 1$ means that present and future actions have the same importance in reaching the final goal.

1.3.5 VALUE FUNCTION

- The agent needs to know how good it is to be in a certain *state*, for deciding which is the proper action to take in that instant. A way of measuring this is given by the *value function* $V_\pi(s)$, defined as the expected sum of rewards that the agent will receive while following a specific policy π from a state s:

$$V_\pi(s) = \mathbb{E}_\pi(R_t \mid s_t = s) = \mathbb{E}_\pi\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid s_t = s\right)$$

This concept can be generalized considering the *action value function*, also known as *Q-function*. It differs from the previous in the fact that it also considers the action. It is defined as the expected sum of rewards while taking an action a in a state s and, therefore, following a policy π:

$$Q_\pi(s,a) = \mathbb{E}_\pi(R_t \mid s_t = s, a_t = a) = \mathbb{E}_\pi\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid s_t = s, a_t = a\right)$$

- A policy π is considered better than another policy π' if the expected return of that policy is greater than π' for all the states s. Therefore, the *optimal value function* $V_*(s)$ can be defined as:

$$V_*(s) = \max_\pi V_\pi(s) \quad \forall s$$

Similarly, the *optimal action value function* $Q_*(s)$ can be defined as:

$$Q_*(s,a) = \max_\pi Q_\pi(s,a) \quad \forall s \quad \forall a$$

- These are called *Bellman Optimality Equations*. When the transition probabilities from a state s to a state s' with an action a are known, it is possible to solve these equations iteratively and define an *optimal policy* π^*:

$$\pi_*(s) = \text{argmax}_a Q_*(s,a)$$

- An agent that chooses its actions by applying the *argmax* operator on *q-values* is said to act greedily.

1.3.6 EXPLORATION VS. EXPLOITATION

- When training a Reinforcement Learning agent, a dilemma often arises: should the agent exploit the acquired knowledge (using the argmax function for selecting the best policy), or should it be free to explore new unknown states for a possible new better policy? Following a known high-reward path can lead to good results but can cause at the same time the possible missing of even better unexplored solutions.

Balancing these two opposite behaviours is fundamental for improving the learning performance. A solution (used in this work) is changing the behaviour of the agent during the learning phase: when the training starts, the agent is required to explore the largest number of states. Moving forward, it starts to exploit the gathered knowledge by focusing on those actions that seems to lead to higher rewards.

- A popular simple yet efficient policy that features such trade-off is the ε - *greedy policy*: the agent acts randomly with a probability ε and greedily with a probability $(1 - \varepsilon)$ where $0 \leq \varepsilon \leq 1$.

ε is the *exploration* parameter of the policy. To change the behaviour of the agent during the training and obtain that process that converges towards a greedy evaluation (optimal policy), a *decay mechanism* of the exploration rate over the training episodes is typically introduced. This should foster exploration in the beginning phases and, as the training progresses, favour exploitation of the optimal policy.

1.3.7 ON POLICY & OFF POLICY EVALUATION

- The methods discussed above are example of *on-policy* evaluations: the q-values of the policy are evaluated by acting under the policy itself. Therefore, the exploration-exploitation trade-off and a decaying scheme towards a greedy policy must be designed as a part of the policy.

- Another approach is the use of *off-policy* methods, which involve two policies: the target policy π, that is learned, and a behaviour policy b which selects the actions during training. This set-up allows the agent to learn a target policy while maintaining an exploration-exploitation trade-off via the behavioural policy.

A noteworthy advantage of off-policy strategies is that they allow the agent to learn the target policy off-line, on data collected under another behaviour policy. A popular consequence is the possibility to use experience *replay buffers* (a memory in which the agent stores samples relative to the current state, action taken, new state, reward).

1.3.8 DEEP RL APPROACH

- When dealing with large number of state-action pairs, using an artificial neural network to approximate Q-function instead of a tabular representation is more memory-efficient. Furthermore, it gives the possibility to interpolate for state-actions pairs that have not been visited during training, which is of fundamental importance to solve problems with high-dimensional continuous observation spaces. The combination of Reinforcement Learning with artificial neural networks results in Deep Reinforcement Learning.

- One of the most common value-based algorithms developed for this purpose is the Deep Q-Learning, which uses a Deep Q-Network (DQN) as function approximator to solve high-dimensional state space problems, but this is only able to handle low-dimensional action spaces. An alternative is using Actor-Critic methods, such as the Deep Deterministic Policy Gradient (DDPG).

1.3.9 ACTOR-CRITIC METHODS

- The DQN algorithm is a *Critic-only* method that rely on an indirect policy representation. These agents are also referred to as value-based, and they use an approximator to represent the action-value function (Q-function). When dealing with continuous space cases, this iterative process can become very expensive from a computational standpoint.

- To solve this problem with continuous spaces, *Actor-Critic* methods have been developed. In these agents, during training, the actor learns the best action to take using feedback from the critic (instead of using the reward directly). At the same time, the critic learns the value function from the rewards so that it can properly update the policy. In general, these agents can handle well both discrete and continuous action spaces.

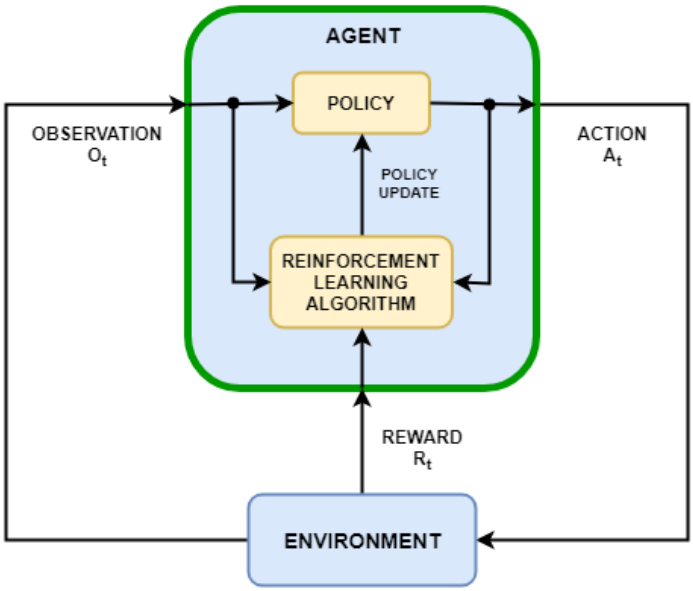


Fig. 4: Schematic representation of a timestep in an Actor-Critic agent

The picture shows the structure of an Actor-Critic agent: the critic is the algorithm that reads reward, observation, action, and criticizes the actor (modifying the policy).

1.4 USED ALGORITHMS

Many Reinforcement learning algorithms have been developed. This chapter introduces the three mainly used in this thesis work:

1.4.1 Q-LEARNING

- *Q-Learning* is a popular example of a critic only, off-policy agent. It belongs to the temporal-learning category of methods because it estimates the return over a 1-step period. The observation space can be continuous or discrete, while the action space is discrete. It basically is a practical implementation of the Bellman Equations, where the agent trains a critic to estimate the future rewards.

- The target policy is greedy, namely selects the actions associated with the highest q-values with the argmax function seen before:

$$\pi_*(s) = \text{argmax}_a Q_*(s, a)$$

While the behaviour policy implements the exploration-exploitation trade-off.

- Here, as opposed to the on-policy cases, there is no need to add a decay in the exploration rate to make the behaviour policy become greedy (though this is possible if needed), since there is already the target policy selecting the best action-value function.

The update formula for the q-values becomes:

$$Q_N^\pi(s_t, a_t) = Q_{N-1}^\pi(s_t, a_t) + \alpha \left(r_{t+1} + \gamma \max_a Q_{N-1}^\pi(s_{t+1}, a) - Q_{N-1}^\pi(s_t, a_t) \right)$$

The data are stored explicitly in a tabular setting, which makes this solution not recommendable for continuous or large state spaces.

1.4.2 DEEP Q-LEARNING (DQN)

- To overcome the limitations of Q-Learning, it is possible to combine it with function approximation, that makes it possible to apply the algorithm to continuous and heavier problems.

- *Deep Q-Learning* (*DQN)* is a critic only, off-policy agent: it is a Q-Learning variant that allows to solve also high-dimensional state spaces. Instead of storing data in a lookup table, this solution uses an adapted artificial neural network as function approximator, speeding up the learning phase generalizing earlier encountered situations to new, unseen, states.

1.4.3 DEEP DETERMINISTIC POLICY GRADIENT

- *Policy gradient (PG)* is a particular way to update the policy by following the gradient itself. *Deterministic policy gradient (DPG)* is an extension of standard PG theorems to deterministic policies.

- *Deep Deterministic Policy Gradient (DDPG)* is a DPG algorithm combined with the advantages of deep neural networks that 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.

If the optimal action-value function $Q_*(s, a)$ is known, then it is possible to find the optimal policy (and action) $\pi_*(s)$ by solving:

$$\pi_*(s) = \text{argmax}_a Q_*(s, a)$$

The difference with DQN relies on the fact that in addition to learning an approximator to $Q_*(s, a)$, DDPG learns also an approximator to $\pi_*(s)$ in a way specifically adapted for environments with continuous action-spaces, by calculating the $\max_a Q_*(s, a)$ in a different way:

- when the action space is discrete, the algorithm just computes the Q-values for each action separately and directly compare them, immediately finding the action that maximizes the Q-values;
- when, instead, the action space is continuous, it becomes impossible to exhaustively calculate and evaluate the space, since the algorithm would need to be run for every little variation of action in the environment, making the process extremely long.

DDPG solves that problem by taking advantage of the fact that if the action space is continuous, then the function the $Q_*(s, a)$ is differentiable. This allows to set up an efficient gradient-based learning rule for a policy $\mu(s)$ which exploits this fact, and then use this to approximate the maximum of the action-value function:

$$\max_a \mathrm{Q}(s, a) \approx \mathrm{Q}\big(s, \mu(s)\big)$$

That translates into the possibility to test any output value in a time efficient manner.

1.5 RL FOR CONTROL APPLICATIONS

- Control of complex industrial systems, with hundreds of variables or more, require an intensive human calibration and optimization, a lot of data, and a lot of time. The objective of this thesis is investigating whether Reinforcement Learning can be a good technique to simplify those situations or not.

- The behaviour of a Reinforcement Learning policy, i.e., how the policy observes the environment and chooses the actions to complete a task in an optimal manner, is not much different from the operation of a traditional feedback controller in a control system.

The following map shows analogies and differences between the two approaches:

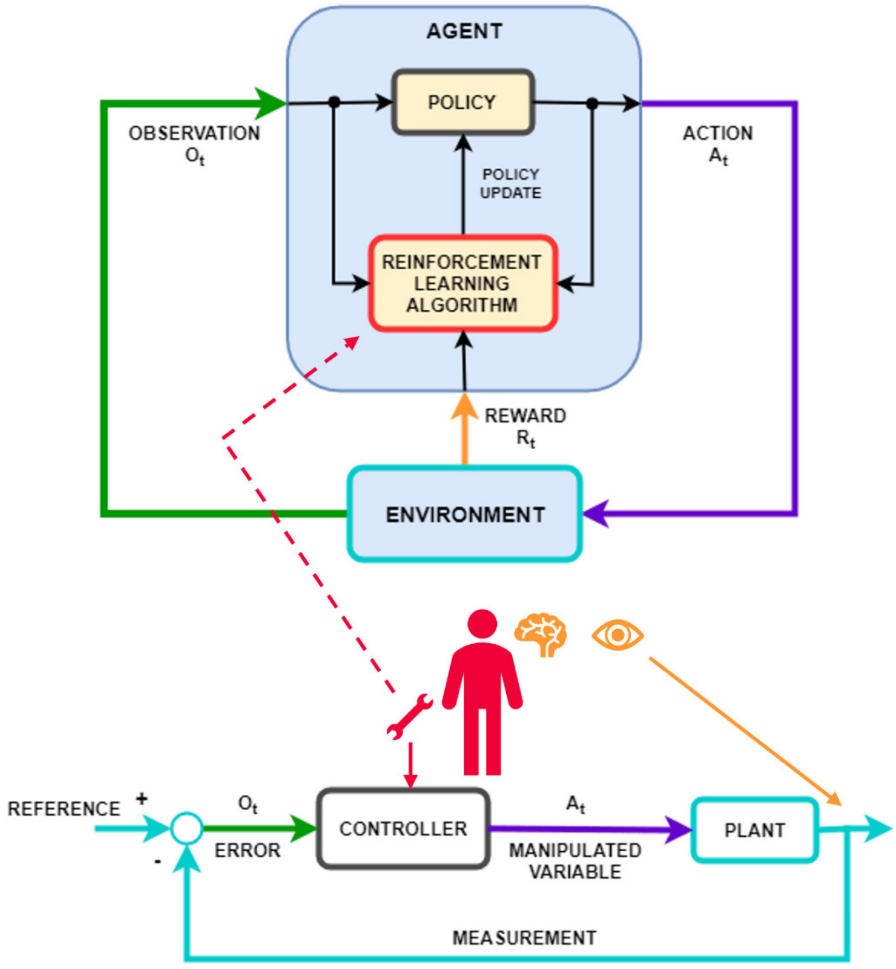


Fig. 5: Comparison of the controller-environment interaction on a Reinforcement Learning agent and on a traditional feedback control system.

In this configuration:

- the *policy*, or the active part of the Reinforcement Learning agent, accomplishes the same function of the controller in a feedback loop;
- the *environment* represents everything is not the controller in the scheme below, including the plant, the reference signal, disturbance signals and other elements;
- the *observation* is comparable to any measurable value visible to the controller, which is the only error signal in PIDs, but could include also the reference, measurement, etc.;
- the *reward* in Reinforcement Learning is what tells the agent (so the controller) how good or bad the system is behaving. In classic controllers this is a human task: a person observes the effect of the controller's action and evaluates possible changes to its parameters, such as the proportional, integrative or derivative gain of a PID;
- the *learning algorithm* can be related to a potential adaptation mechanism of an adaptive controller or, as before, a variation of the parameters of a PID.

Many control problems, in addition to having many parameters, can also be particularly difficult to tune in terms of gains and parameters, requiring the experience of a control engineer. Deep Reinforcement Learning, on the other hand, provide a way to use a self-taught agent that can eventually be deployed very efficiently on the final system. This is opening up many new opportunities for specific data-intensive uses and for those cases where actions need to be generated directly from raw data inputs, such as images or video (which is especially useful in sectors like automated driving, defence, supervision, etc.).

2. SETUP

2.1 SIMCENTER AMESIM

- *Simcenter Amesim* is a 1D simulation platform for modelling and analysing multi-domain mechatronic systems. It includes a suite of tools to model, analyse, and predict the behaviour of hydraulic, pneumatic, thermal, electric, or mechanical systems. It also features a set of libraries that contain pre-defined components for different physical domains.

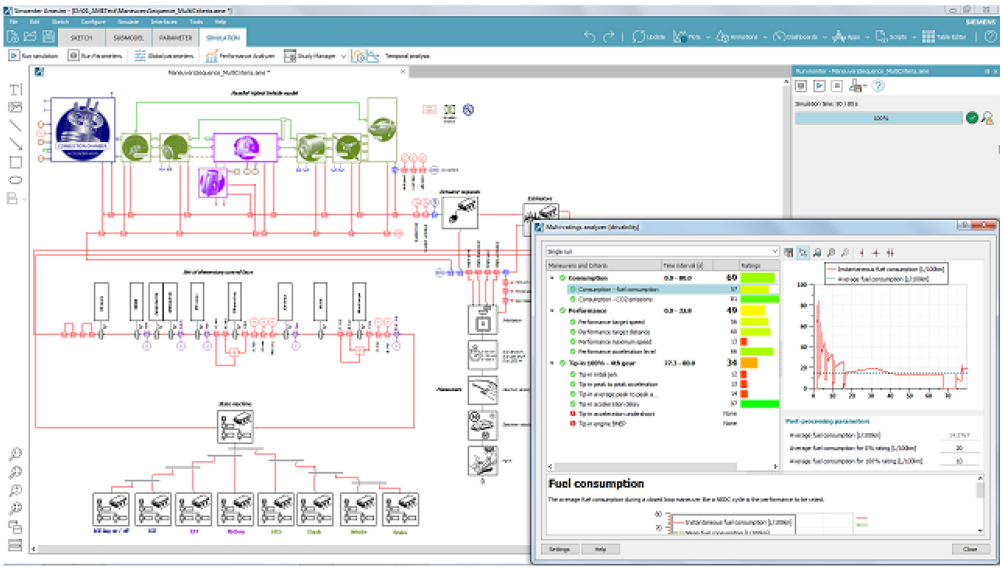


Fig. 6: Simcenter Amesim UI

- Amesim is used to prepare the PID-controlled physical models of the various analysed systems.

- Later, the control part is complemented (or replaced) with an *interface block* that allows an external software (in this case, Simcenter Studio) to take the control of the model, that will indeed become the *environment* of the Reinforcement Learning agent.

2.2 SIMCENTER STUDIO

- *Simcenter Studio* is a generative engineering software that allows generating and evaluating system architectures on early concept phase. It presents a notebook-based UI that allows to combine the programming code with plot and text, which is something that makes this program particularly suitable for interactive presentations.

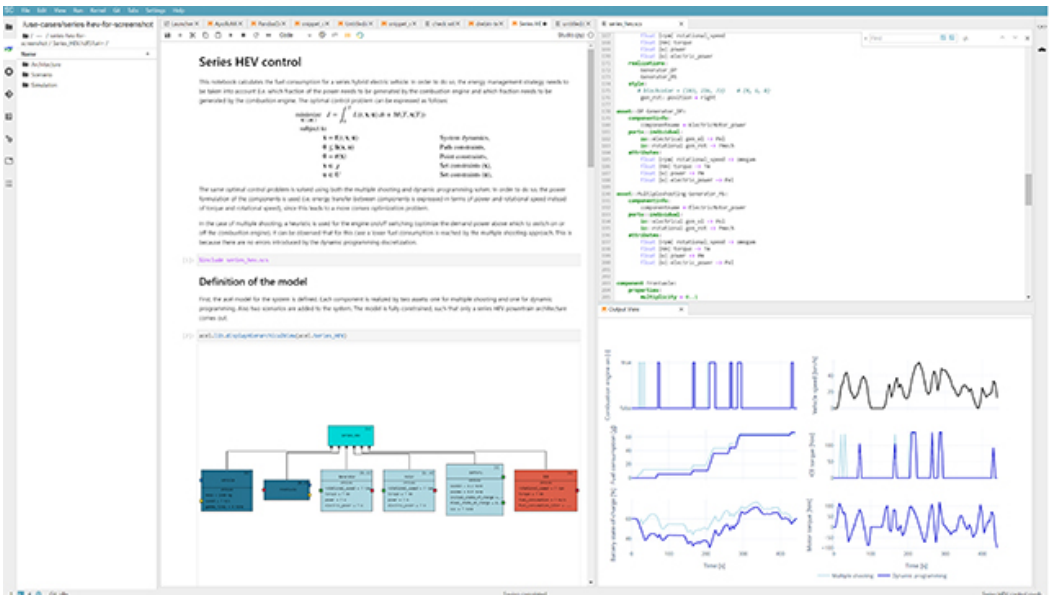


Fig. 7: Simcenter Studio UI

- With the help of several included libraries, it is possible to run various Machine Learning and Reinforcement Learning algorithms to analyse data or run simulation. The environments can be either downloaded from public online collections (such as the *OpenAI Gym*) or uploaded by the user. In this work, Studio is used to:

- Write Python code to define the Reinforcement Learning algorithms and hyperparameters;
- Run co-simulations of the algorithms on the custom environment models uploaded from Amesim;
- Plot the physical effects of the deployed policy on various parameters of the model (such as position, velocity, acceleration, etc.).

2.3 WORKFLOW

- The goal of the thesis is to investigate and evaluate the applicability of Reinforcement Learning for control applications. To do so, simple examples are initially analysed, slowly moving towards more complex mechanisms. Initially, some tests on different control environments have been done, some directly taken from the OpenAI Gym, some manually modelled on Amesim. This thesis will discuss the results of two different environments: an Inverted Pendulum on Cart and an Electromechanical Elevator.

- The necessary steps for train a Reinforcement Learning agent are the following:

1. *Formulation of the problem*: definition of the tasks and the goals the agent must achieve;
2. *Preparation of the environment*: 1D modelling of the system, addition of the interface block on Amesim to control the model with Studio;
3. *Definition of the rewards*: writing of how the reward is measured from the agent to evaluate its performance and modify the policy. Correct balancing of multiple rewards, if present, by assigning them a different weight.
4. *Creation of the agent*: configuration of the agent learning algorithm, discretization (if required), setting of the hyperparameters (values that control the learning process during the training);
5. *Training of the agent*: run the simulation with the chosen agent on the environment with the defined rewards;
6. *Evaluation of the agent*: simulation of the deployment best policy from the training on the environment and verification of the performances.

Note that the training of an agent isn't always straightforward: it can happen, and it does, that the processes don't initially converge to a good enough policy; in this case it will be necessary to modify one of the steps listed above and restart the procedure.

3. EXPERIMENTAL TESTS

3.1 INVERTED PENDULUM ON CART

3.1.1 SYSTEM MODEL

The first case study is the Inverted Pendulum on Cart: an initial PID controlled model of the pendulum is considered:

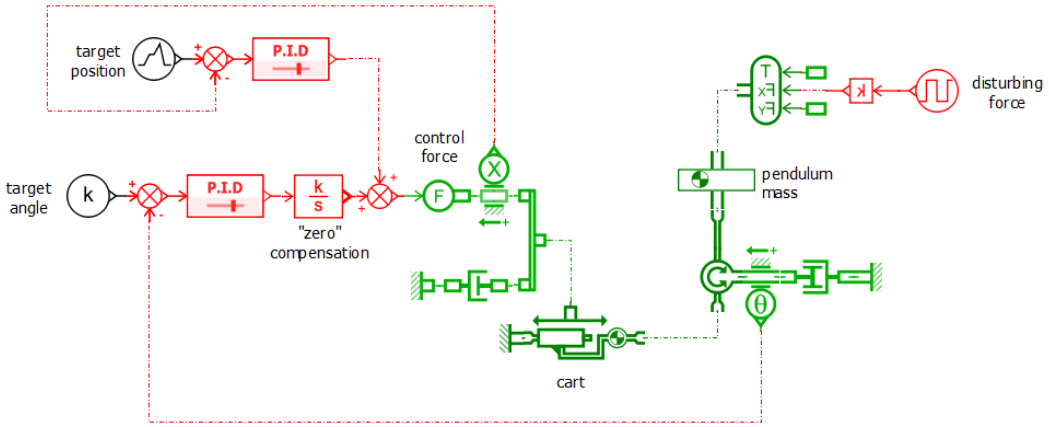


Fig. 8: PID controlled inverted pendulum – 1D Amesim model

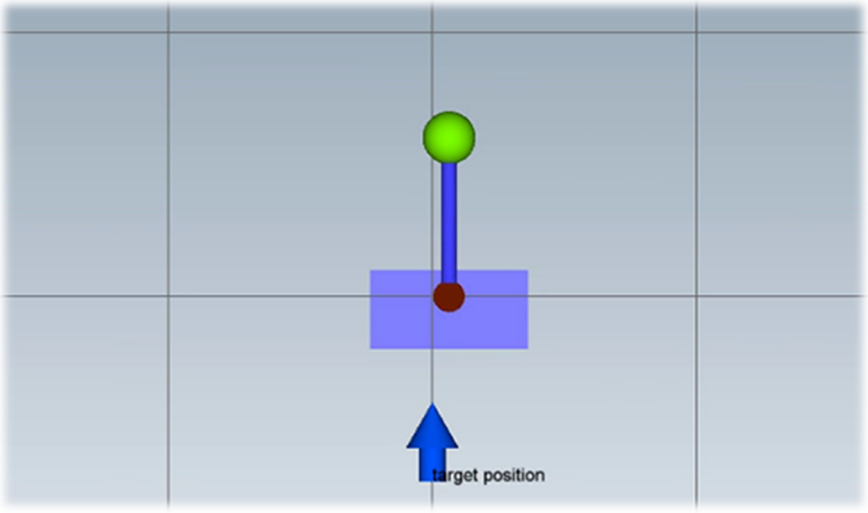


Fig. 9: PID controlled inverted pendulum – screenshot of a 2D Amesim simulation with moving target position

- The first picture shows the 1D scheme of the pendulum: there is a cart, constrained to move horizontally, a pole with a mass on top of it, and two PIDs: one is designated to follow a horizontal moving target position, while the other controls the cart to maintain the pole in vertical position. On the other side there is a disturb: this is a 20N force applied every few seconds on the mass.

- The second picture is a screenshot of a 2D simulation of the system: the cart can follow sufficiently well the target maintaining the pole in vertical position, but there is a small position error that accumulates over time. So, the question now is: can a Reinforcement Learning agent help to solve this problem?

To test a RL agent, the block diagram is modified to add an interface block in parallel to the PID controllers, that it is controlled from Studio. The functional scheme is the same, except that now the feedback signals go to the PID and into the interface. The control force is the sum of the PID action and the output of the block.

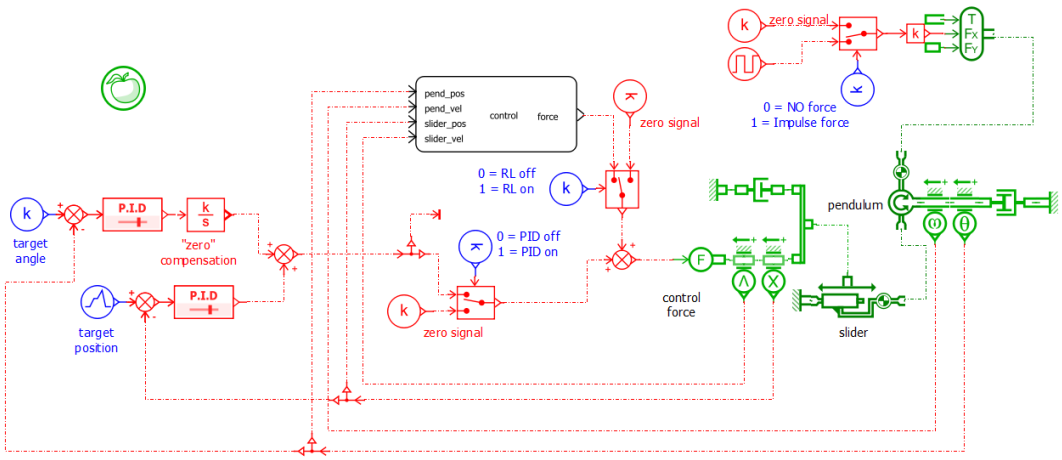


Fig. 10: PID + RL controlled inverted pendulum – 1D Amesim model

- As visible from the picture, also some switches have been added, to give the possibility of including or excluding the PID signal and/or the agent signal selectively. This means there are three possible configurations:

- PID on, RL off – the same of the initial model;
- PID off, RL on – only reinforcement learning;
- PID on, RL on – combined solution called *Residual Reinforcement Learning*.

- The idea is to upload the model on Studio and train a Reinforcement Learning agent to control the cart via the interface block, in addition and in replacement of the PID, for then comparing the performances of the three different configurations.

3.1.2 Q-LEARNING CO-SIMULATION

- The training and the evaluation of the agent take place on Studio, where a Python notebook is configured.

```
class PendulumOnCartEnv(PendulumOnCart):
    def define(self):
        PendulumOnCart.define(self)
        self.data.update({'theta_max': 0.1, 'x_max': 0.5})

    def evaluateReward(self):
        o_tp1 = self.getOutputSpace().getValues_tp1()
        if self.data['time'] >= self.data['T']-self.communication_time:
            end_episode = True
        elif abs(o_tp1[2])>self.data['x_max'] or abs(o_tp1[0]-np.pi)>self.data['theta_max']:
            end_episode = True
        else:
            end_episode = False
        r_tp1 = 1.
        terminal_state = False
        return r_tp1, terminal_state, end_episode

    def shapeReward(self, r_tp1, terminal_state, end_episode):
        o_tp1 = self.getOutputSpace().getValues_tp1()
        if abs(o_tp1[2])>self.data['x_max'] or abs(o_tp1[0]-np.pi)>self.data['theta_max']:
            r_tp1 = -1.
        else:
            o_t = self.getOutputSpace().getValues_t()
            r_tp1 = int(abs(o_t[0]-np.pi) >= abs(o_tp1[0]-np.pi))
        return r_tp1
pendulum_on_cart_env = PendulumOnCartEnv(pool_name)
```

The agent will be be trained with the Q-learning algorithm. Because of the tabular representation, a discretization of the continuous environment outputs is necessary. In addition, the policy is forced to output values from a list.

```
import scs_reinforce.agents as scs_agents

options = scs_agents.QLearning.getOptions()
options.discount_rate = discount_rate
options.exploration.decay_exploration_rate = 10**(decay_exploration_rate_exp[qq])
options.tabular_values.decay_learning_rate = 10**(decay_learning_rate_exp[rr])

agent = options.createAgent()

from scs_reinforce.spaces import DiscretizeContinuousSpace, ListAction

input_shaping = [ DiscretizeContinuousSpace(n_div=[8, 8, 8, 8],
                    min_values=[np.pi-0.1, -2.5e-1, -1.5e-1, -2.5e-1],
                    max_values=[np.pi+0.1, 2.5e-1, 1.5e-1, 2.5e-1]) ]

agent.setInputShaping(input_shaping)
output_shaping = [ ListAction([[-5.], [0.], [5.]]) ]
agent.setOutputShaping(output_shaping)
```

Fig. 11: Definition of the environment and of the agent hyperparameters

- Following the required steps for training a RL agent, the notebook is configured for the simulation by setting the environment properties, including:

- the model physical parameters (lengths, mass, inertia);
- which control to use (PID/RL/both);
- the initial conditions of each training episode (a tiny random angle).

- After that, the reward is defined: in this case, a very simple discrete function is used, giving to the agent a +1 point for each timestep on which horizontal position of the cart and the pendulum angle remain within a specific defined range.

- Lastly, the Q-Learning hyperparameters (discount rate, exploration rate, learning rate, decay of the exploration rate, decay of the learning rate) and the discretization of the input and output space (required with Q-Learning) are set.

3.1.3 COMPARISON BETWEEN PID AND RL CONTROL

- After training, the results can be plotted:

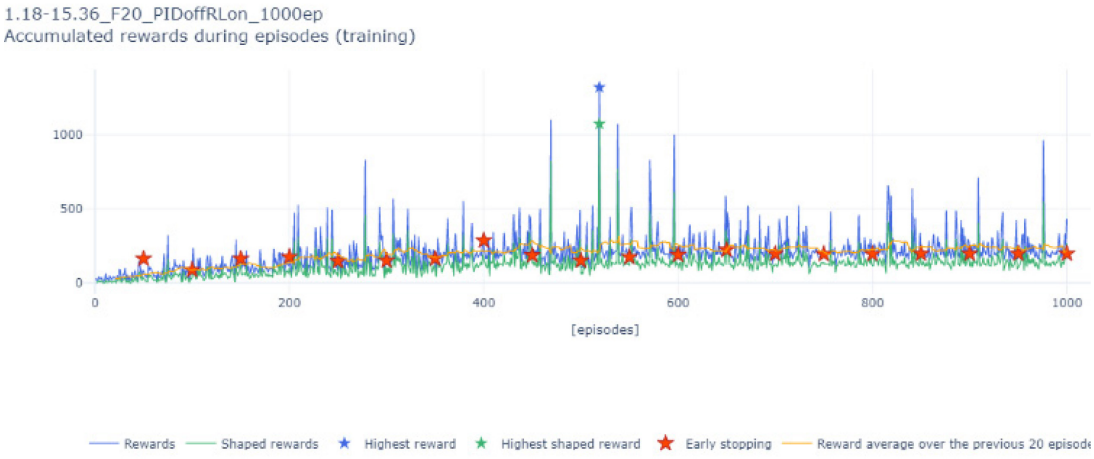


Fig. 12: Accumulated rewards over 1000 training episodes – RL agent only

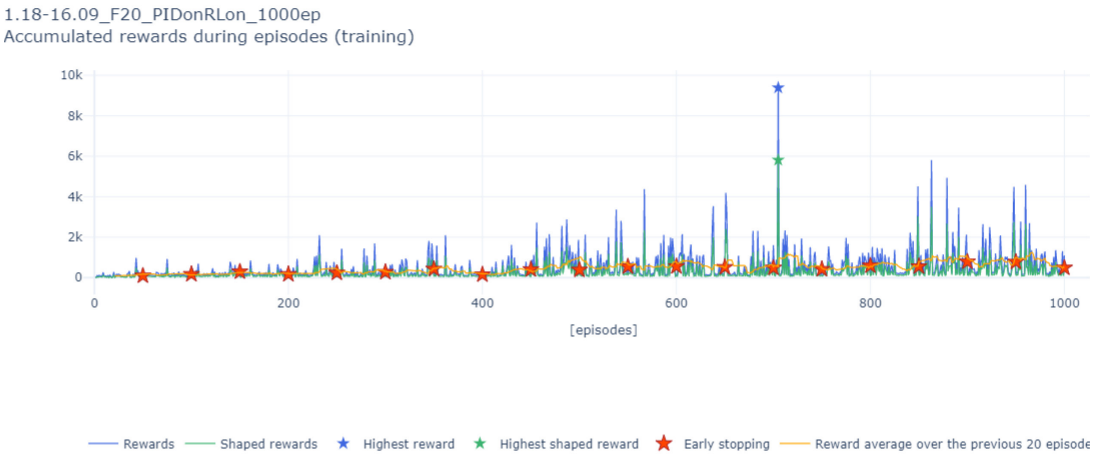


Fig. 13: Accumulated rewards over 1000 training episodes – PID + RL agent

- The graphs are showing the accumulated rewards for each episode and an evaluation of the best policy found during the process (red star). Above, the PID control is turned off via the switch, and the Reinforcement Learning agent must control the system by itself (through the interface block). Below, both the PID and the agent are connected (Residual Reinforcement Learning). It is clear that the second situation, when the PID is helping, gives higher rewards. The PID-only case accumulated rewards are not shown here because it would not make sense, given that with the output RL signal from the interface block disconnected from the system the results of the training are never applied on the environment.

- The next step is the evaluation of the best policy on 10 random episodes:

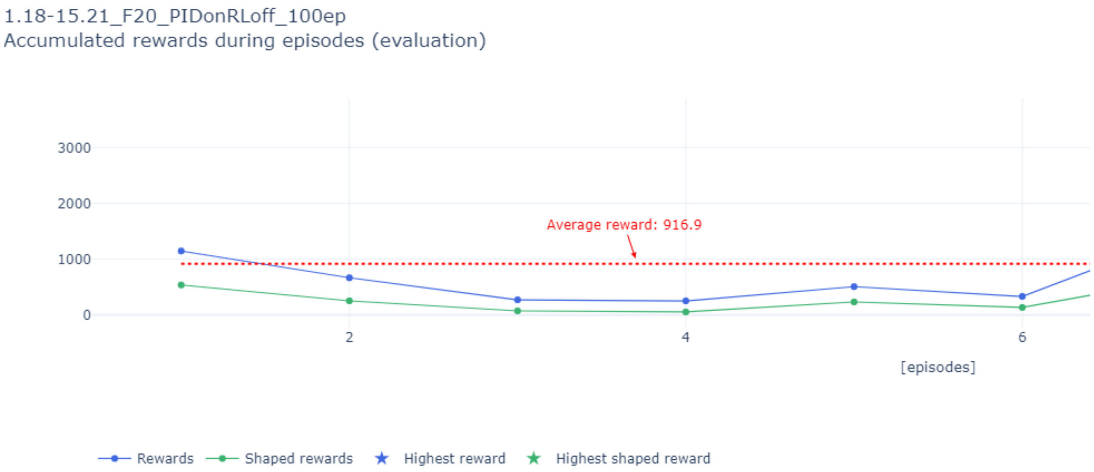


Fig. 14: Accumulated rewards over 10 evaluation episodes – PID only

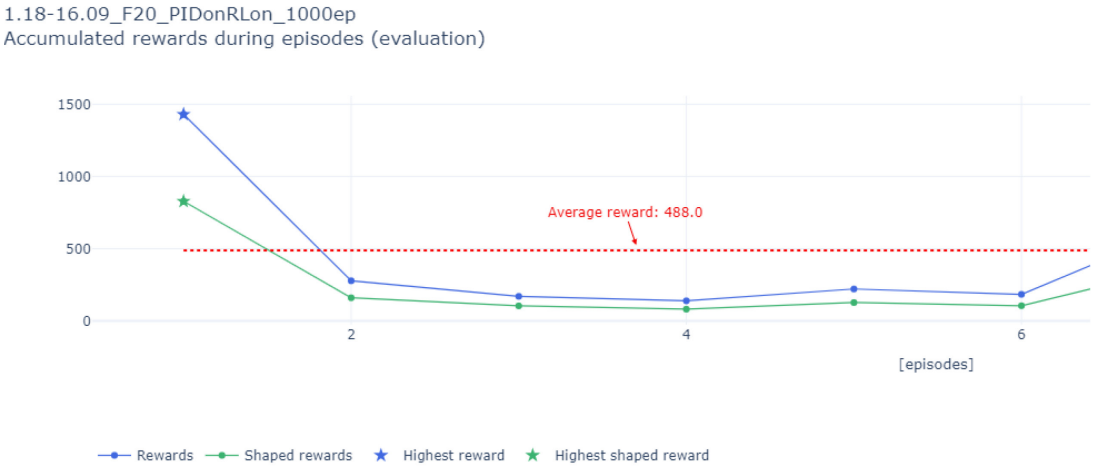


Fig. 15: Accumulated rewards over 10 evaluation episodes – PID + RL agent

- The first graph shows the rewards for the PID only case, the second for the residual Reinforcement Learning case, both representing the evaluation of the best policy on 10 random episodes. The red line indicates the average reward over 10 episodes. Here, on the contrary, it makes sense to plot the results of the PID only case for the comparison, because even though the learned policy is not applied (the interface block is disconnected), it is still possible to read the average reward got from the system in reaction to the PID control and compare this with the residual Reinforcement Learning case (when the interface block is connected and the RL agent helping in control).

- It seems, however, that when the RL agent is inserted the rewards are lower, meaning that instead of helping in reaching the goal the agent is making the system behaving worse. The cause might be related or to an excessively sparse discretization, or to a wrong choice of the hyperparameters.

To see if that is the case, some additional simulations are done (always on the residual Reinforcement Learning setting), leaving the same initial values of Exploration Rate and Learning Rate but changing the decay rate of these during the episodes:

- Initial Exploration rate = 1.0
- Decay Exploration rate = 10^ [(-3.0), (-2.5), (-2.0), (-1.5), (-1.0)]
- Initial Learning rate = 0.5
- Decay Learning rate = 10^ [(-4.0), (-3.5), (-3.0), (-2.5), (-2.0)]

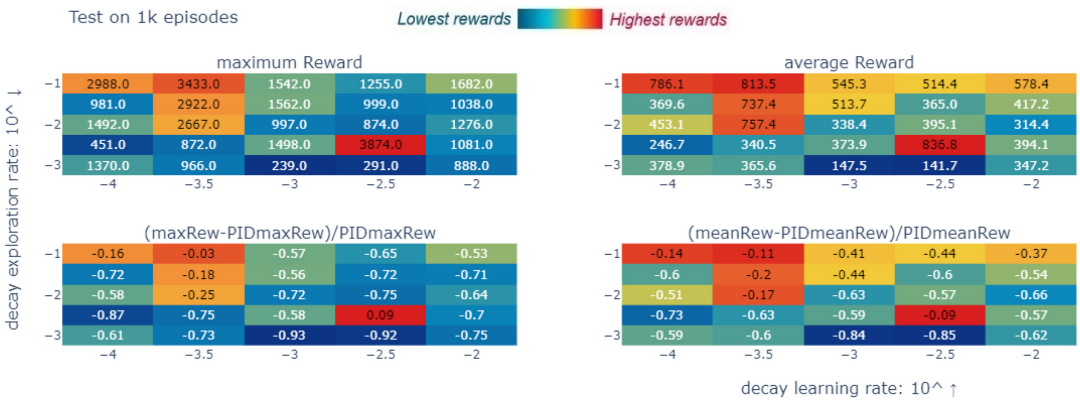


Fig. 16: Maximum and Average accumulated rewards over 10 evaluation episodes – PID + RL agent – changing the exploration rate and learning rate decay

- A training for every combination of the decay values is launched in series, for a total of 25 simulations. Each policy is evaluated, and the maximum and average rewards obtained out of the ten evaluation episodes plotted in the first line of the six heatmaps, where the x-axis represents a change of the learning rate decay and the y-axis a change of the exploration rate decay. In the second line, there are the same plots divided by the rewards of the PID only case, which gives a "specific" value indicating how near or far the performance of each combination is from the one of the PID only situation.

It seems from the maps that there isn't any clear magic value, but some give indeed better results than other (the reddish areas), though the up-left area specifically gives higher rewards than the initial values (which are in the middle of each map). Nevertheless, it is always slightly worse than how it was with the PID only scenario.

- To evaluate the actual physical effects of the policies on the pendulum, the state and the action spaces of one evaluation episode is plotted for both PID only and PID+RL cases:

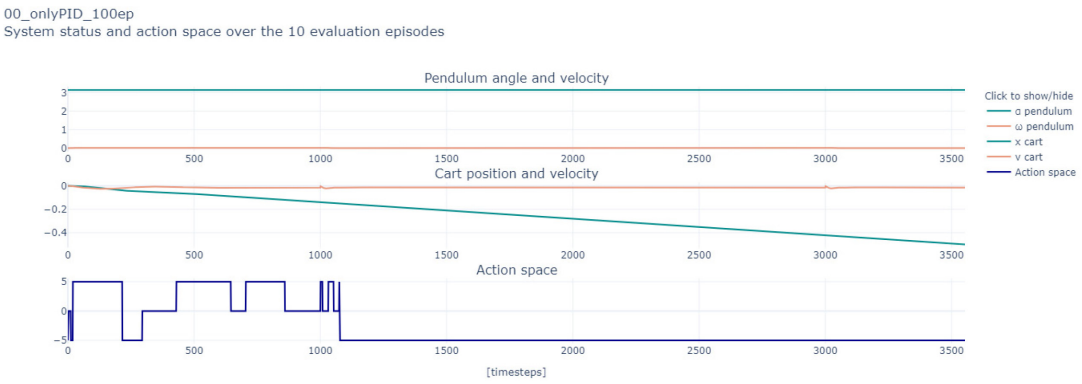


Fig. 17: State and Action space of one episode – PID only

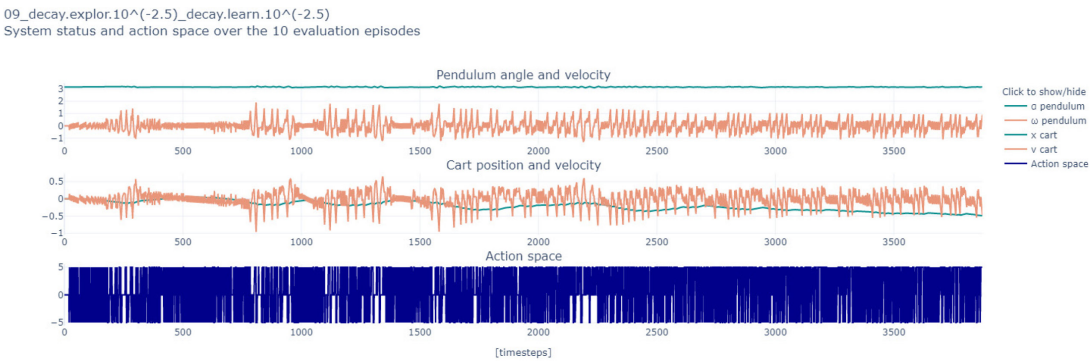


Fig. 18: State and Action space of one episode – PID +RL agent

The green line represents the position (angle of the pendulum in the upper plot, distance of the cart from the setpoint in the central plot), the orange line the velocity (angular velocity of the pendulum in the upper plot, speed of the cart in the central plot), and the blue line is the action, which is the force applied on the cart from the agent (discretized according to the initial agent settings). It is to be noted again that the interface block (so the agent) output is not connected to the cart in the PID only simulations, so the blue line on the plots above doesn't have any effect.

- The outcome, in any case, is clear: the cart is in both cases sliding in one direction, but the PID controlled system is much more stable and linear than the residual Reinforcement Learning control, which on the other hand makes the pendulum oscillating more. This is due to the fact that the Q-Learning agent utilizes a discretized input and output space.

Results would probably improve a lot with either a more densely discretised input and output space (at the expense, of course, of longer simulation time) or using a continuous action space agent (such as DDPG). However, these agents are not tested on the inverted pendulum on cart for this work.

3.2 ELECTROMECHANICAL ELEVATOR

3.2.1 SYSTEM MODEL

- The second case study is the Electromechanical Elevator: this one is only controlled through the interface block:

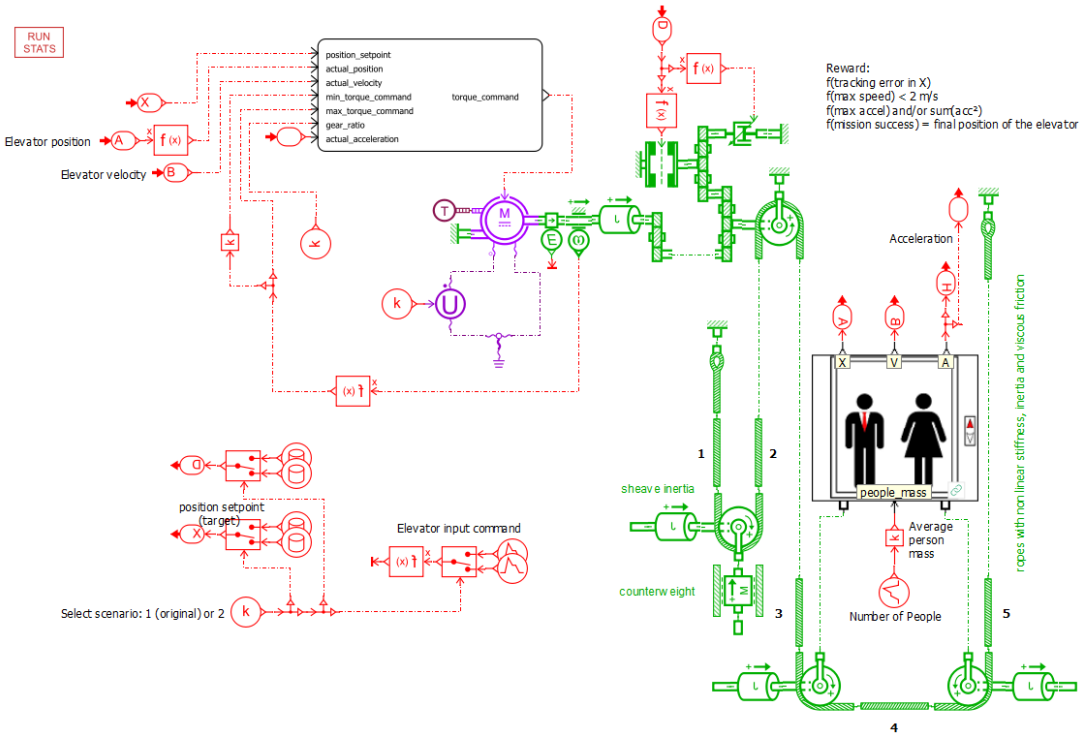


Fig. 19: RL controlled electromechanical elevator – 1D Amesim model

- The model can be divided in two parts:

- on the left (red) there is the control section with the interface block: this presents seven inputs: position setpoint, actual position, actual velocity, actual acceleration, minimum torque, maximum torque, gear ratio – and one output: torque command (which goes to the electrical motor).
- on the right (green) there is the mechanical model of the elevator: an electrical motor connected to a shaft that rotates a series of gears and pulleys that move the elevator.

Fig. 20: Electromechanical elevator – screenshot of a 3D Amesim simulation while following a five-step trajectory

- The position setpoint given to the elevator is a five-steps trajectory between the floors 0 → 4 → 0 → 1 → 3. On each stop the mass of the elevator is different, simulating a different number of people on board, so the agent must also adjust the torque accordingly.

3.2.2 DQN CO-SIMULATION – TRAJECTORY

- The training and the evaluation of the agent are done on Studio, where a Python notebook is configured.

```
import numpy as np
from scs_reinforce.environments import EnvironmentAmesimCosim

class Elevator_class(EnvironmentAmesimCosim):
    file_name = 'Elevator01'
    communication_time = 0.1
    data = {
        'T': 70.,      # final time of system's simulation
       }

    def reset(self):
        return np.array([0., 0., 0., 0., 0., 0.])

    def evaluateReward(self):
        o_tp1 = self.state_space.getValues_tp1()
        if self.data['time'] >= self.data['T'] - self.communication_time:
            end_episode = True
        else:
            end_episode = False
        terminal_state = False

        delta = round(10*abs(o_tp1[0] - o_tp1[1]))
        r_tp1 = -delta
#         if abs(o_tp1[6]) > 1.0:
#             r_tp1 += -5
#         deltaacc = round(abs(o_tp1[6] - o_t[6]))

        return r_tp1, terminal_state, end_episode
elevator_class = Elevator_class(pool_name)
```

Set output discretization

```
from scs_reinforce.spaces import ListAction
import numpy as np

qwe = np.arange(torque_min, torque_max+1, torque_discretization)
asd = []
for i in range(len(qwe)):
    asd.append([qwe[i]])
print('list_actions: ' + str(asd))

list_actions = ListAction(asd)
elevator_class.setInputShaping([list_actions])

from scs_reinforce.spaces import StackSteps, SelectSpaceIndex
step_stacker = StackSteps(depth=2)
space_index_selector = SelectSpaceIndex([0,1,2,3,4,5,8,9])

elevator_class.setOutputShaping([step_stacker, space_index_selector])
```

```
list_actions: [[-50.0], [-40.0], [-30.0], [-20.0], [-10.0], [0.0], [10.0], [20.0], [30.0], [40.0], [50.0]]
```

Fig. 21: Definition of the environment and of the agent hyperparameters

- Following the required steps for training a RL agent, the notebook is configured for the simulation by setting the environment properties, which in this case are always the same initial conditions from the Amesim model.

- After that, the reward is defined: for this DQN test, a continuous reward is used, which consists of a negative score given to the agent proportional to the elevator actual gap on each timestep from the trajectory setpoint. In other words, the agent must follow the setpoint and the farther it is, the lower the score it gets.

- Lastly, the hyperparameters (discount rate, exploration rate, learning rate, decay of the exploration rate, decay of the learning rate) are set.

With DQN it is not necessary to discretize the input space. The discretization of the output space (torque) consists of actions between -50 Nm and +50 Nm divided by steps of 10 ([-50, -40, -30, -20, -10, 0, +10, +20, +30, +40, +50]).

- For this initial test, 200 episodes are run.

Fig. 22: Accumulated rewards over 200 training episodes – DQN agent

The rewards are always negative, obviously, because it is only penalized the distance of the elevator from its setpoint. However, the agent has a good learning trajectory.

- The best policy is found during the episode 150. It can be useful to see how the physical model is actually behaving in this circumstance by plotting the state space (trajectory, velocity and the acceleration of the elevator) and the action space (torque signal):

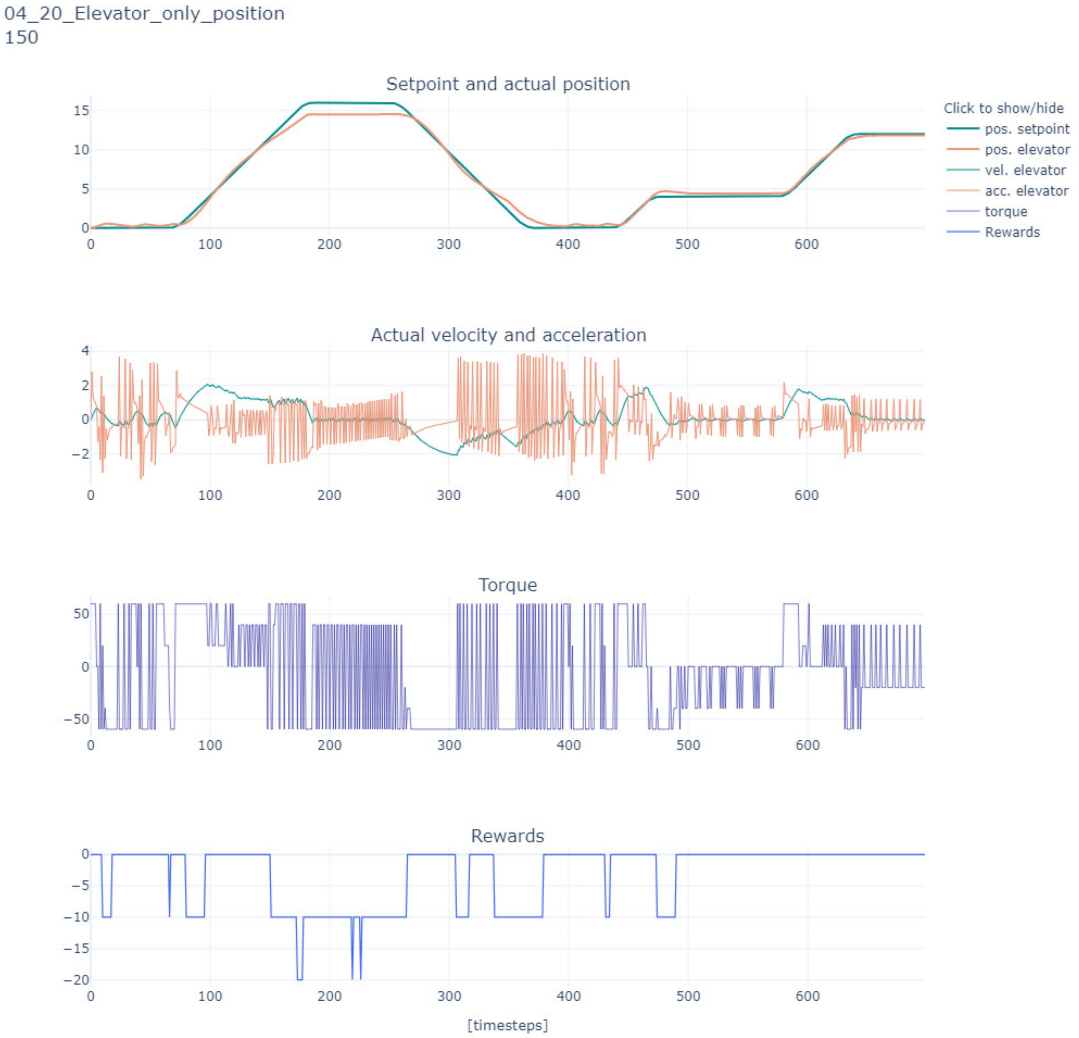


Fig. 23: Trajectory, velocity, acceleration, torque, rewards of the best policy – DQN agent

The first plot from the top shows the elevator's position setpoint (green) and current (orange). The second the velocity (green) and the acceleration (orange). The third the torque signal from the agent to the system and the fourth the rewards accumulated by the agent at each timestep.

The rewards are often around zero, meaning that the policy is able to follow the established trajectory in a very good way.

The only two visible problem here are: a steady positional error on constant phases and some high acceleration peak, which in an elevator would be uncomfortable for the hypothetical people on board.

- To solve these two problems, three approaches are tested:

1. Trying out different hyperparameters combinations;
2. Adding a reward penalizing high accelerations;
3. Using a continuous action space agent (DDPG), that should give less impulsive actions than a discrete output space agent.

- Starting from the first of the three solutions, the same approach of the pendulum case is followed: keep some values fixed, change some others, and try out different combinations to see how they do affect the simulation.

- Initial Exploration rate = [0.50, 0.75, 0.97]
- Decay Exploration rate = 0.0025
- Initial Learning rate = 0.0025
- Output torque list discretization = [5, 10]

As example, the accumulated rewards, trajectory, velocity, acceleration, torque signal plots of two of these combinations are shown in the next pages:

- Torque discretization = 10 ; Initial Exploration rate = 0.97

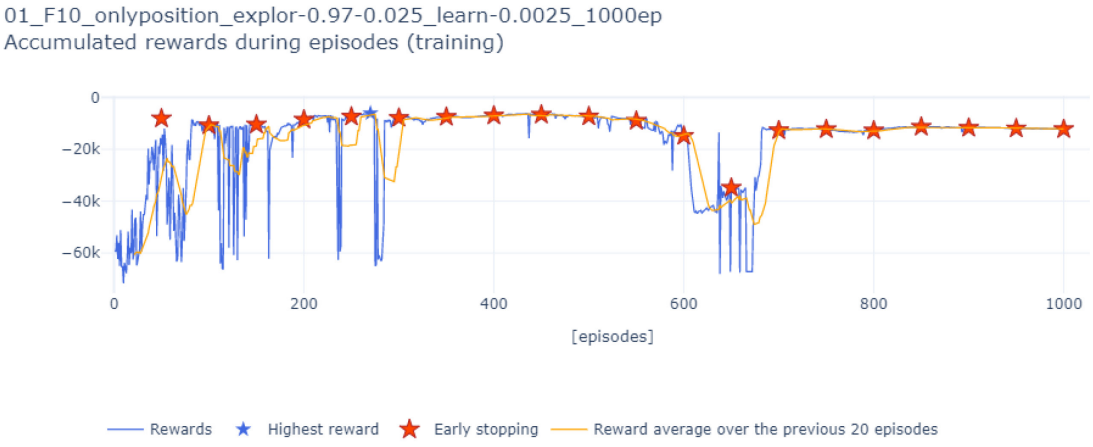


Fig. 24: Accumulated rewards over 1000 training episodes – DQN agent

- Torque discretization = 5 ; Initial Exploration rate = 0.5

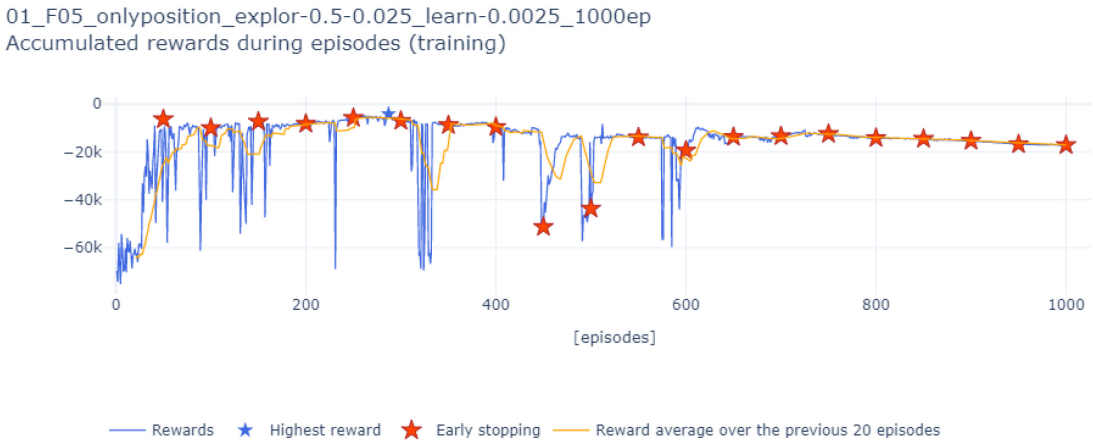
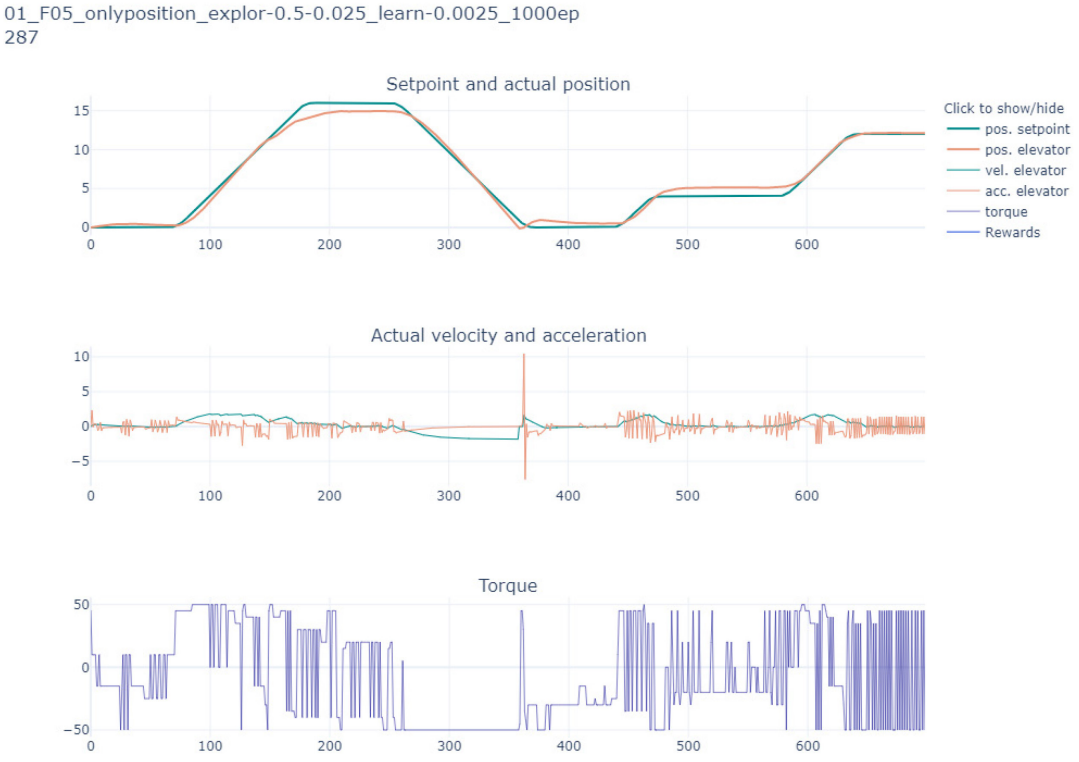


Fig. 25: Accumulated rewards over 1000 training episodes – DQN agent

- The setting with higher torque discretization, which means smaller steps between the output possible actions, give moderately better results, as expected. It is visible also how the exploration rate affects the training process, with the learning curve oscillating much more in the first case when the rate is higher (so the agent is exploring more).

- Torque discretization = 10 ; Initial Exploration rate = 0.97

- Torque discretization = 5 ; Initial Exploration rate = 0.5



- There doesn't seem to be much difference from the initial situation unfortunately: there still is a small position error during the "waiting" phases and some high acceleration peak.

3.2.3 DQN CO-SIMULATION – TRAJECTORY & ACCELERATION

- Moving to the second of the three solutions: adjusting the reward function so that it penalises higher accelerations.

The new reward is, for each timestep, a negative score proportional to the sum of the elevator actual gap from the trajectory setpoint and a value (-10) added when the acceleration > 1.0 m/s².

- The rest (environment, trajectory, hyperparameters, torque discretization) remains the same, to allow a comparison of the effects of the different reward function alone.

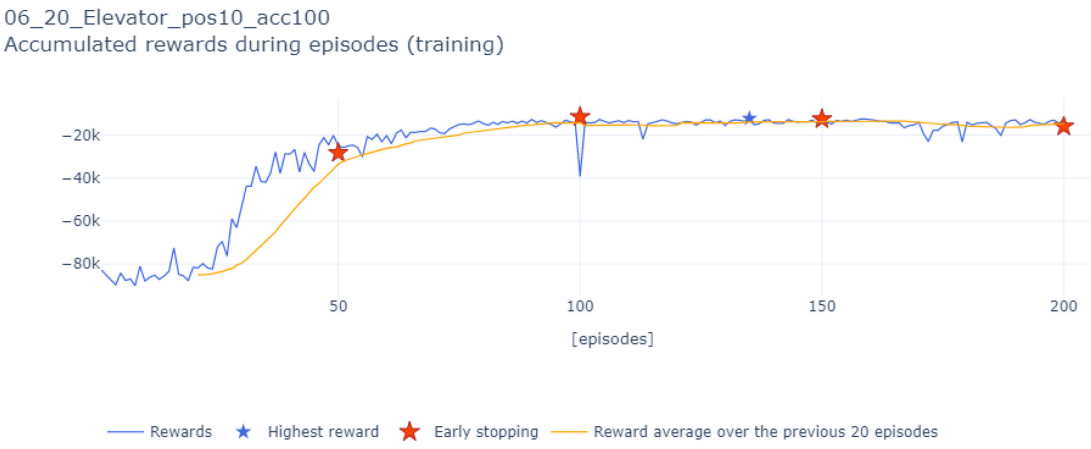


Fig. 28: Accumulated rewards over 200 training episodes – DQN agent

The rewards are much lower than before because there is the (negative) addition of the acceleration penalty.

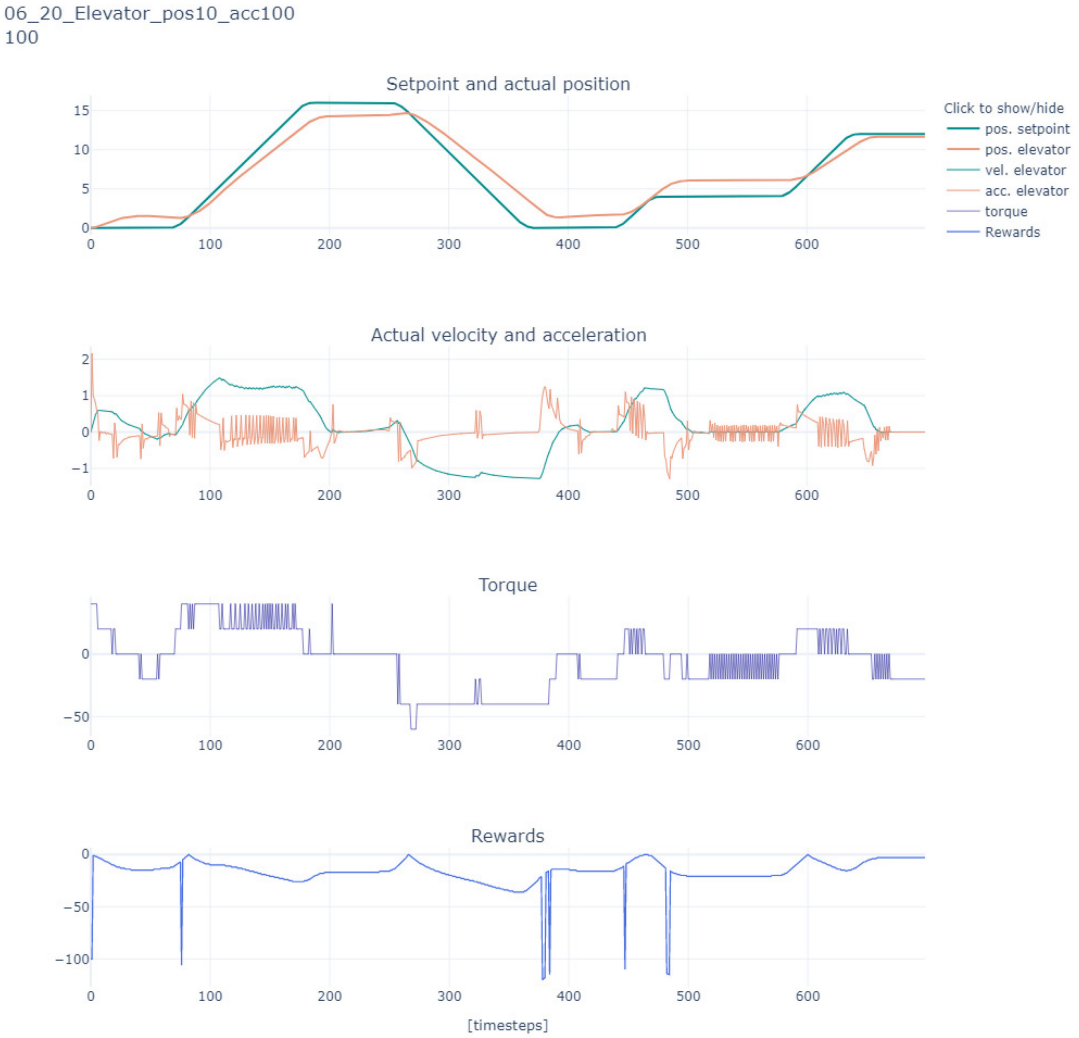


Fig. 29: Trajectory, velocity, acceleration, torque, rewards of the best policy – DQN agent

The state space shows that the accelerations are indeed almost always within the not-penalized range, but this comes at the expense of the rapidity to reach the desired position. This is something expected, since the two summed rewards are mutually exclusive (a quick reaction to the set positions changes require a high acceleration, and vice versa). The more the acceleration is penalized, the more "damped" the trajectory graph will appear.

3.2.4 DDPG CO-SIMULATION – TRAJECTORY

- Finally, the third of the three solutions: using a continuous action space agent (DDPG).

```
import numpy as np
from scs_reinforce.environments import EnvironmentAmesimCosim

class Elevator_class(EnvironmentAmesimCosim):
    file_name = 'Elevator01'
    communication_time = 0.1

    data = {
#         'g': 9.81,   # gravity acceleration
        'T': 70.,      # final time of system's simulation
        }

    def reset(self):
        return np.array([0., 0., 0., 0., 0., 0., 0.])

    def evaluateReward(self):
#         o_t
        o_tp1 = self.state_space.getValues_tp1()

        if self.data['time'] >= self.data['T'] - self.communication_time:
            end_episode = True
        else:
            end_episode = False

        terminal_state = False

        delta = (10*abs(o_tp1[0] - o_tp1[1]))
        r_tp1 = -delta

#         if abs(o_tp1[6]) > 1.0:
#             r_tp1 += -5
#         deltaacc = round(abs(o_tp1[6] - o_t[6]))

        return r_tp1, terminal_state, end_episode

elevator_class = Elevator_class(pool_name)
```

```
from scs_reinforce.spaces import ContinuousSpace, ShapingDef

class scaleInputs(ShapingDef):
    def define(self):
        self.shaping_space = ContinuousSpace(1)

    @staticmethod
    def shapingMethod(instance, space_values):
        torque = space_values
#         print(space_values, type(space_values))
        asdf = np.array([space_values[0]*50])
#         print(asdf, type(asdf))
        return  asdf

input_shaping_env = [ scaleInputs() ]
elevator_class.setInputShaping(input_shaping_env)
```

Fig. 30: Definition of the environment and of the agent hyperparameters

- A continuous reward is applied, which consists of a negative score given to the agent proportional to the elevator actual gap on each timestep from the trajectory setpoint (the same of the first case, without penalizing the acceleration).

- The output torque is always between -50 Nm and +50 Nm, but continuous. The following DDPG hyperparameters are initially used:

- Initial Noise level = 0.1
- Decay Noise level = 0.0001
- Scaling Action = 1.0
- Actor Learning rate = 0.0005
- Critic Learning rate = 0.0010
- Hidden Layers = (12-24-48-24)

The noise level has conceptually the same meaning of to allow a comparison of the effects of the different reward function alone.

Fig. 31: Accumulated rewards over 500 training episodes – DDPG agent

Accumulated rewards are very high at the beginning, but they fall back down after not many training episodes. This behaviour is probably a consequence of a phenomenon, already well known in scientific literature, called "catastrophic forgetting", which is caused by some combinations of weights between neurons in the hidden layers that can lead to bad result and completely overwrite previously learned information upon learning new information. This is not completely solved but the situation will much improve later, when hyperparameters will be changed.

- The early stoppings (red stars), which are used as the policy evaluation parameters, are very low, meaning that the agent is not learning anything. A plot of a state space of one evaluation is shown below to check what happens on the model:

Fig. 32: Trajectory, velocity, acceleration, torque, rewards of one policy – DDPG agent

The agent is indeed not learning, but always giving the maximum torque as output (here indicated by 1, which is then multiplied by 50, the maximum set value).

- To understand what is causing that behaviour, and if it is possible to get better policies, different hyperparameters combinations will be tried, following the same process already seen in the previous cases.

There are a total of six changeable hyperparameters here. Since it is impossible to visualize a six-dimension map to find the best combinations, several simulations will be launched, keeping 3 of them fixed and changing the other 3, in an iterative optimization process divided in multiple parts:

Changing hyperparameters, part 1

- Keeping fixed the values of:

  - Hidden Layers = (12-24-48-24)
  - Actor Learning rate = 0.0005
  - Critic Learning rate = 0.0010

- Trainings with different combinations of the following are launched:

  - Scaling Action = [+1.00, +1.78, +3.16, +5.62, +9.77]
  - Initial Noise level = 10^ [(-2.0), (-1.5), (-1.0), (-0.5), (+0.0)]
  - Decay Noise level = 10^ [(-5.0), (-4.5), (-4.0), (-3.5), (-3.0)]

This means a total of 5x5x5 = 125 different simulations are done, and the results are divided in five heatmaps, to see in an iterative way what are the combination of hyperparameters that improve the learning process of the agent and lead to better policies.

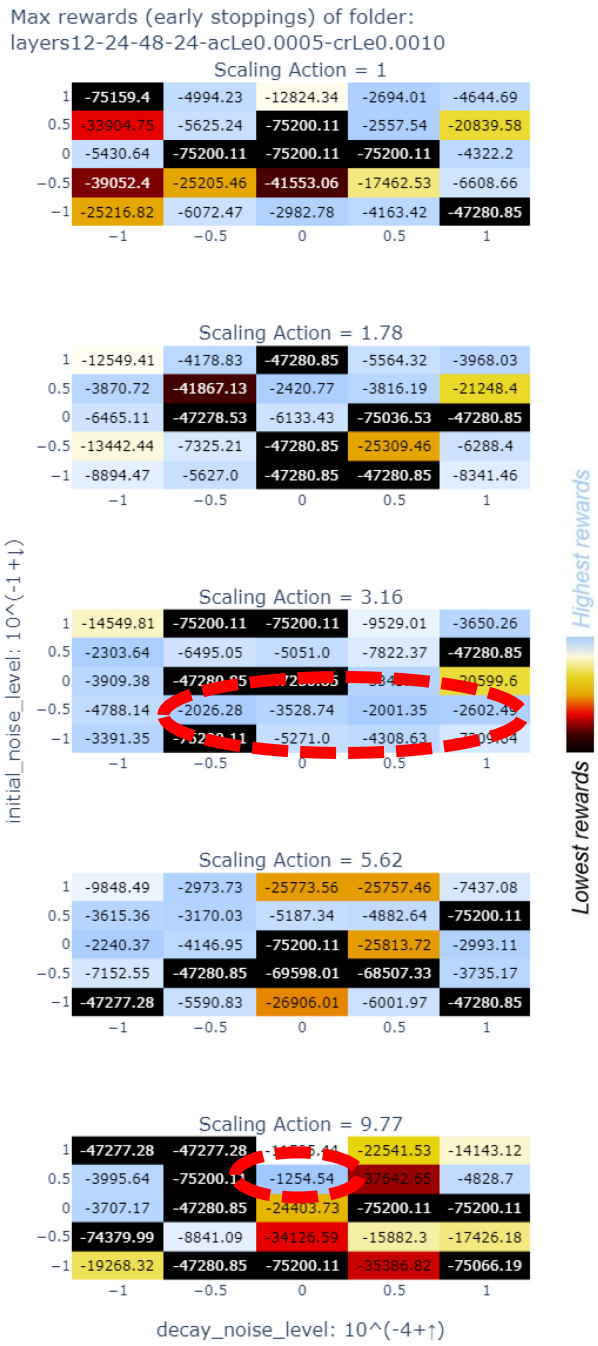


Fig. 33: Changing hyperparameters, part 1 - Max accumulated rewards

There isn't a clear trend of which scaling actions or noise is better to choose for getting better policies, but it is possible to identify some areas of combinations (circled in red) that seems to get very high accumulated rewards.

- Continuing with this iterative process, the other three -previously fixed- hyperparameters are now getting changed, keeping the scaling action, initial noise level, and decay of the noise level fixed with the values that gave the better result on the previous tests (the two conditions circled in red):

Changing hyperparameters, part 2	Changing hyperparameters, part 3
- Keeping fixed the values of: • Scaling Action = 3.16 • Initial Noise level = 10^(-4.0) • Decay Noise level = 10^(-4.0)	- Keeping fixed the values of: • Scaling Action = 9.77 • Initial Noise level = 10^(-0.5) • Decay Noise level = 10^(-4.0)

- Trainings with different combinations of the following are launched:

  - Hidden Layers = [(12-24-12), (24-48-24), (6-12-24-12), (12-24-48-24), (6-12-24-48-24-12)]
  - Actor Learning rate = 10^ [(-4.0), (-3.5), (-3.0), (-2.5), (-2.0)]
  - Critic Learning rate = 10^ [(-4.0), (-3.5), (-3.0), (-2.5), (-2.0)]

With the same procedure, simulations will be launched for each combination of fixed values above. So divided in five heatmaps with a total of 125 simulations for the combination in part 2, and another five heatmaps with a total of 125 simulations for the combination in part 3.

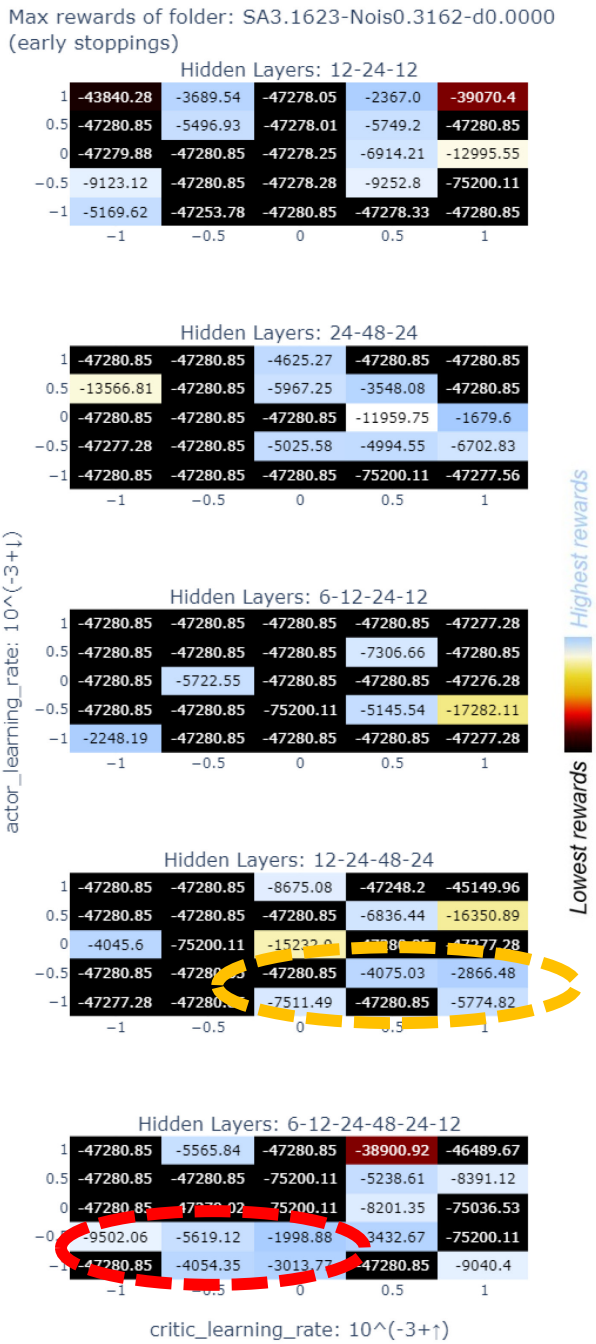


Fig. 34: Changing hyperparameters, part 2 - Max accumulated rewards

The circles indicate the higher rewards; the yellow is the initial combination of hidden layers, which has already been tested. It might be interesting however to later evaluate the red one.

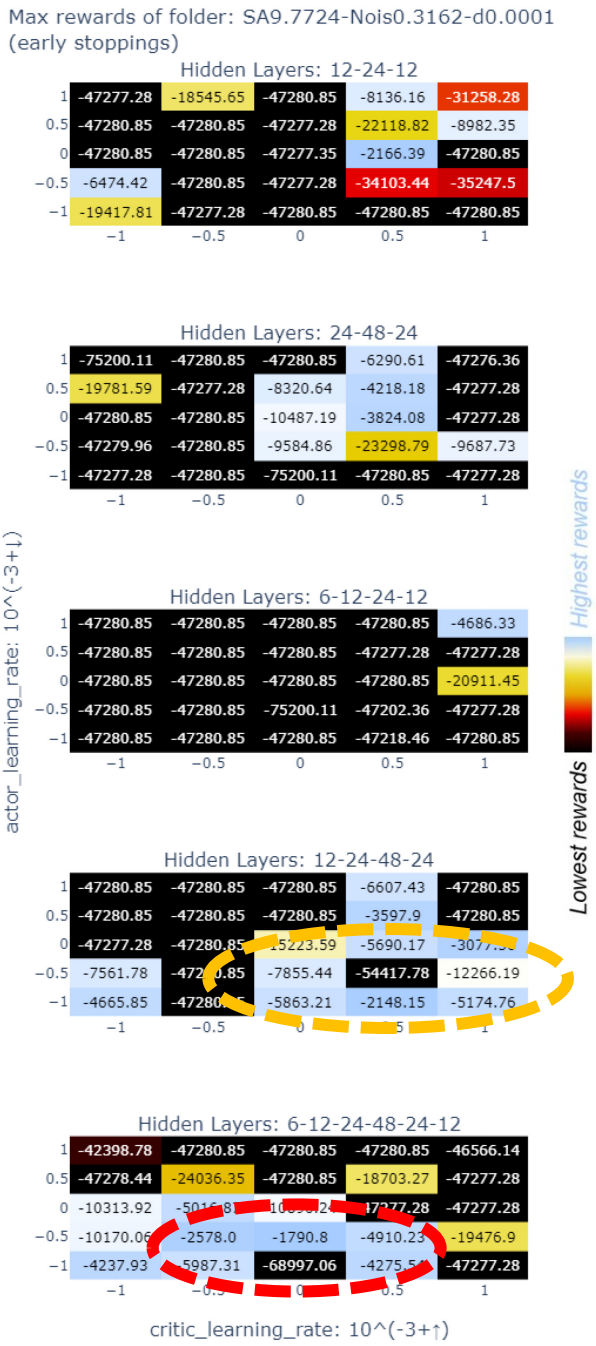


Fig. 35: Changing hyperparameters, part 3 - Max accumulated rewards

As before, the more the hidden layers the better the results. The yellow layer's setting (12-24-48-24) has been tested already, so the red hidden layer combination (6-12-24-48-24-12) can be now finally tested.

- A final test is done, where the bigger neural network among the tested so far is used in combination with the actor learning rate and the critic learning rate that gave the best results in the previous simulations. The scaling action, initial noise level and decay of the noise level are changing among the same combinations tested initially:

Changing hyperparameters, part 4

- Keeping fixed the values of:

  - Hidden Layers = (6-12-24-48-24-12)
  - Actor Learning rate = 0.0005
  - Critic Learning rate = 0.0010

- Trainings with different combinations of the following are launched:

  - Scaling Action = [+1.00, +1.78, +3.16, +5.62, +9.77]
  - Initial Noise level = 10^ [(-2.0), (-1.5), (-1.0), (-0.5), (+0.0)]
  - Decay Noise level = 10^ [(-5.0), (-4.5), (-4.0), (-3.5), (-3.0)]

The plots of these 125 simulations are again divided in five heatmaps.

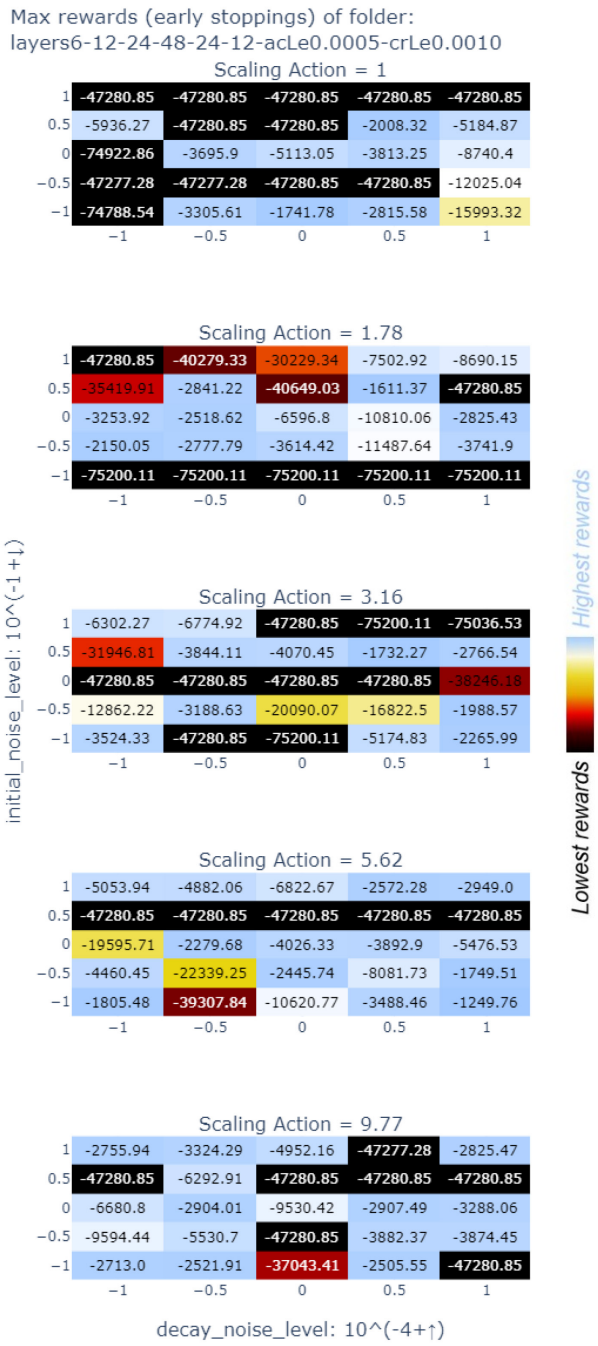


Fig. 36: Changing hyperparameters, part 4 - Max accumulated rewards

Here the accumulated rewards are in general higher compared to the previous cases, which is a sign that a higher number of hidden layers leads facilitate the learning process. Nonetheless, there is still a strong dependency on the other variables.

- Despite the long series of iterative tests on the different combinations, the DDPG rewards were almost always higher than the previous cases.

Just as example, the state space and action space related to a combination of hyperparameters that resulted in a particularly high rewards are shown below:

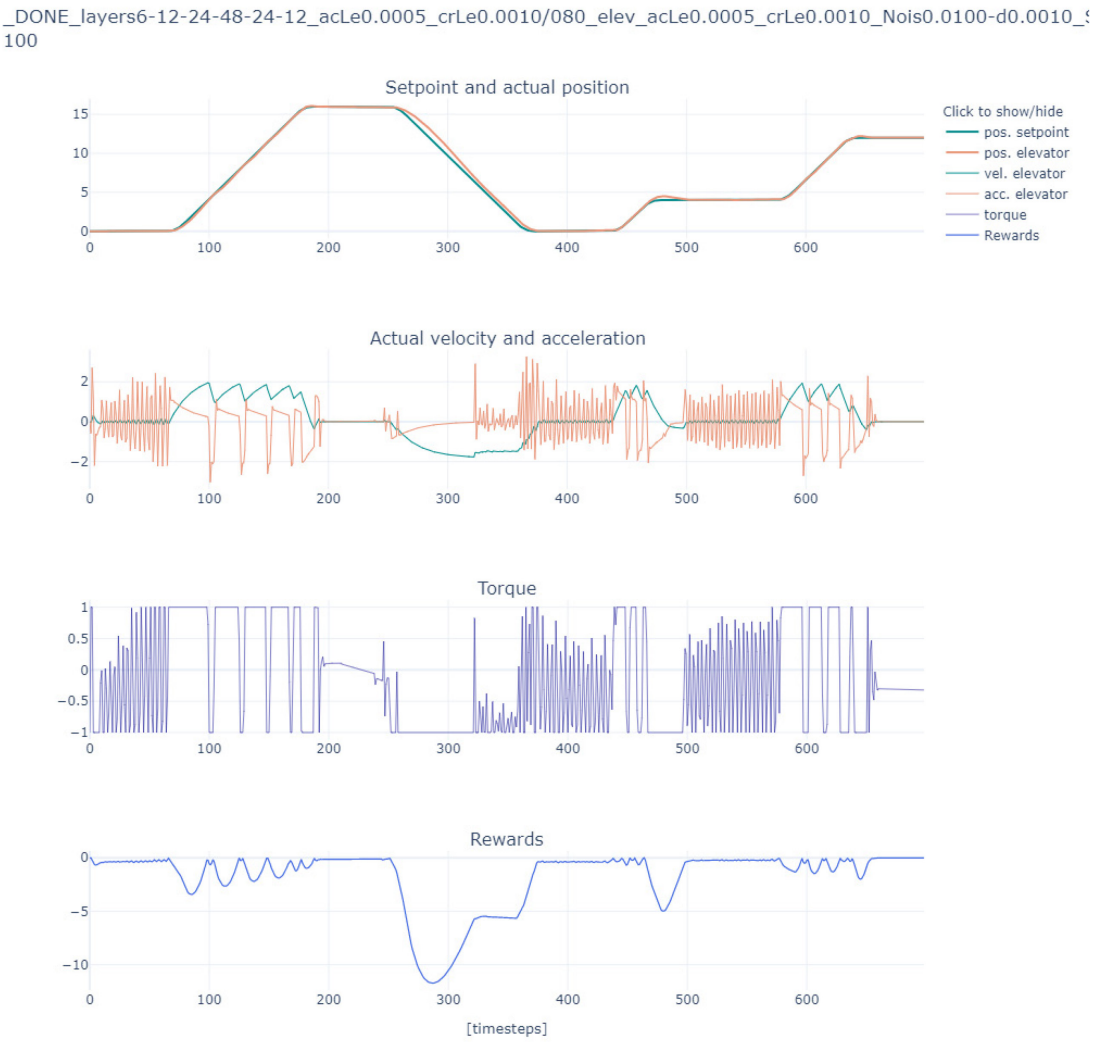


Fig. 37: Trajectory, velocity, acceleration, torque, rewards of one good policy – DDPG agent

The trajectory following is practically perfect and there are not relevant acceleration peaks (still a bit high sometimes, but these can be reduced only smoothing the trajectory setpoint).

4. CONCLUSIONS

4.1 KEY TAKEAWAYS

- The main objective of this thesis work, made in collaboration with Siemens Digital Industry Software, was trying to understand the viability of using Reinforcement Learning techniques for the control of mechatronic systems by simulating different physical scenarios using the software Simcenter Amesim in interoperability with Simcenter Studio. In this regard, the most important takeaway is that not only it is feasible, but it is also possible to reach good results.

Contrary to what someone might expect, however, it is not an easy and straightforward process where one can just throw the model and the indications of the desired task to some AI algorithm, hoping that it will figure out the rest. Machine Learning is a very complex subject and plenty of variables can affect the result, especially when artificial neural networks are involved. An example is the catastrophic forgetting phenomenon analysed during the initial DDPG trainings.

Therefore, human intervention is still fundamental, aside from the preparation of the models, for choosing the correct algorithms and the related hyperparameters on the basis of the task, the model environment and, no less important, the available computational power and training time. Choosing the best trade off out of these requirements still requires human expertise, at least at the current state of the art of this technology.

- It was also seen that, in general, using continuous rewards (incremental or decremental) instead of discrete/boolean ones (like +1 or -1) and using continuous input and output space agents improves the agent's learning performance a lot. At the expenses, again, of longer training times and more computational power needed.

- When a system needs to reach at the same time multiple mutually exclusive goals (such as trajectory following and acceleration limiting on the elevator example), the related contributions on the reward function must be thoroughly weighted, until the best balance among the two (or more) is found. Endless possibilities of functions are deployable, and this is, again, something that requires some expertise.

4.2 WHAT'S NEXT?

- Talking about the elevator case, to go further with the experiments, it would make sense to try giving step signal setpoints instead of trajectory ones and to give multiple setpoints at the same time (as it would be on a real system), to see how the agent manages multiple indications. Also, using different reward functions penalizing, in addition to the acceleration, the wait time and the energy consumption is something that can be done, with the goal of optimizing the floor trajectory the elevator is to follow (indispensable in case of multiple setpoint).

Then, after having made experience on simpler models, the natural next step is moving towards more complicated, multivariable systems.

- Reinforcement Learning is a relatively new technique, but it has the potential of enabling the use of Artificial Intelligence a lot of different sectors. With respect of control applications, the biggest advantage is the possibility of trying out many different solutions and evaluating the performances of different system architectures already during the early concept phase of an engineering system.

As previously said, these operations are not simple, and more advanced optimization procedures are needed in order to speed up the whole process. Nevertheless, by dedicating some time on perfecting the models, the algorithms, and the reward functions, great results can be reached.

5. BIBLIOGRAPHY

BIBLIOGRAPHY

- https://www.plm.automation.siemens.com/global/en/products/simcenter/, [Online; accessed October 2021].

- https://it.mathworks.com/help/reinforcement-learning/, [Online; accessed October 2021].

- https://spinningup.openai.com/en/latest/algorithms/ddpg.html, [Online; accessed October 2021].

- Daniel Luis Simões Marta. *Deep Learning Methods for Reinforcement Learning,* 2016.

- Divyam Rastogi. *Deep Reinforcement Learning for Bipedal Robots*, 2017.

- Cordella Giovanni Marco. *Continuous control of ugv for mapless navigation: a virtual-to-real deep reinforcement learning approach*, 2018.

- Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare and Joelle Pineau. *An Introduction to Deep Reinforcement Learning*, 2018.