DEEP REINFORCEMENT LEARNING ALGORITHMS IN MULTI AGENT CHANGING ENVIRONMENTS USING POTENTIAL FIELDS

A Degree Thesis

Submitted to the Faculty of the

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

by

Joan Lo Anguera

In partial fulfilment

of the requirements for the degree in

Telecommunications Technologies and Services Engineering

Advisor: José Antonio Lázaro

Barcelona, September 2022

Abstract

The project explores the possibilities offered by reinforcement learning in the field of robotics with the vision of guiding robots in changing environments with collision avoidance through potential fields.

For this, the DDPG, TD3, SAC and PPO reinforcement learning algorithms are implemented through the Matlab Toolbox "Reinforcement Learning" with the aim of carrying out a comparative study on which of them is the most optimal for different configurations of environments and parameters, with the help of training graphs and statistical tables.

Also, potential fields have been developed in this project, demonstrating to be a suitable tool for guiding robots in changing environments, and even to implement multi agent scenarios, avoiding collisions among them and enhancing collaboration.

Resum

El projecte explora les possibilitats que ofereix l'aprenentatge per reforç en l'àmbit de la robòtica amb la visió de guiar robots a través d'entorns canviants amb evitació de col·lisions mitjançant camps de potencials.

Per això s'implementen els algorismes d'aprenentatge per reforç DDPG, TD3, SAC i PPO per mitjà de la Toolbox de Matlab Reinforcement Learning amb l'objectiu de fer un estudi comparatiu sobre quin d'ells és el més òptim per a diferents configuracions d'entorns i paràmetres; tot això amb l'ajuda de gràfiques d'entrenament i taules estadístiques.

Així mateix, s'han desenvolupat camps potencials en aquest projecte, demostrant ser una eina adequada per a guiar robots en entorns canviants, i fins i tot per implementar escenaris multiagent, evitant col·lisions entre ells i potenciant la col·laboració.

Resumen

El proyecto explora las posibilidades que ofrece el aprendizaje por refuerzo en el ámbito de la robótica con la visión de guiar a robots a través de entornos cambiantes con evitación de colisiones mediante campos potenciales.

Para ello se implementan los algoritmos de aprendizaje por refuerzo DDPG, TD3, SAC y PPO por intermedio de la Toolbox de Matlab "Reinforcement Learning" con el objetivo de realizar un estudio comparativo sobre cuál de ellos es el más óptimo para diferentes configuraciones de entornos y parámetros; todo ello con la ayuda de gráficas de entrenamiento y tablas estadísticas.

Además, en este proyecto se han desarrollado campos potenciales, demostrando ser una herramienta adecuada para guiar robots en entornos cambiantes, e incluso implementar escenarios multiagente, evitando colisiones entre ellos y potenciando la colaboración.

A mi madre, mi hermana, tíos y abuelos por haberme apoyado siempre, y a todos y cada uno de los compañeros y profesores que me han ayudado y acompañado durante el trayecto de la carrera.

Acknowledgements

I want to thank my tutor José Antonio Lázaro for guiding me during the project. Looking backwards, I could not have imagined the culmination of this work. The help of Joaquín Fernández Piqueras and the IT team has also been invaluable, for his support to access the CALCULA server of the TSC department, to have the necessary resources to carry out this project.

Revision history and approval record

Revision	Date	Purpose
0	10/03/2022	Document creation
1	29/9/2022	Document revision

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Joan Lo Anguera	joan.anguera@estudiantat.upc.edu
José Antonio Lázaro	jose.lazaro@tsc.upc.edu

Written by:		Reviewed and approved by:	
Date	10/03/2022	Date	29/9/2022
Name	Joan Lo Anguera	Name	José Antonio Lázaro
Position	Project Author	Position	Project Supervisor

Table of contents

1 Introduction **12**
1.1 Motivation 12
1.2 Objectives 12
1.3 Artificial Intelligence 13
1.4 Gant diagram, work packages, tasks and milestones 16
2 State of the art of the technology used or applied in this thesis: **17**
2.1 Reinforcement Learning 17
2.1.1 Markov property 18
2.1.2 Markov decision processes (MDP) 19
2.2 Reinforcement Learning Agents 20
2.2.1 Actor-Critic Agents 20
2.2.2 Deep Deterministic Policy Gradient (DDPG) 20
2.2.3 Twin-Delayed Deep Deterministic Policy Gradient (TD3) 21
2.2.4 Proximal Policy Optimization (PPO) 23
2.2.5 Soft-Actor Critic (SAC) 25
2.2.6 Potential Field Algorithm 27
2.3 Neural Networks 28
2.4 Adaptative moment estimation algorithm (Adam) 29
3 Methodology / project development: **30**
3.1 Static Environment 30
3.1.1 Single agent 31
3.1.2 Two agents 33
3.2 Changing environment 33
3.2.1 Single agent 33
3.3 Providing continuous rewards 34
3.3.1 Static environment with a single agent 34
3.3.2 Static environment with two agents 34
3.3.3 Changing environment with a single agent 35
3.4 Neural networks and training 36
4 Results **38**
4.1 Static Environment 38
4.1.1 Single Agent 38
4.1.2 Two Agents 41
4.2 Changing Environment 43

4.2.1 Single agent 43

5 Budget **44**

6 Conclusions and future development **45**

List of Figures

Figure 1: Typical Reinforcement Learning Scenario. 15
Figure 2: Generalised Reinforcement Learning scheme. 17
Figure 3: Action selection of a SAC agent. 25
Figure 4: Different potential field configurations in space between two particles. 27
Figure 5: Neural networks scheme. 28
Figure 6: Perceptron model. 28
Figure 7: Sigmoid, Tanh, ReLU and Softplus activation functions. 29
Figure 8: Static environment and robot forces representation. 30
Figure 9: Initial Simulink model for one agent. 31
Figure 10: Agent and Environment blocks of the initial Simulink model for one agent. 31
Figure 11: Sketch of the Simulink model for two agents. 33
Figure 12: Default reward function. 34
Figure 13: Two robot reward function. 35
Figure 14: Changing environment reward function for one robot. 35
Figure 15: Actor and critic neural networks for DDPG and TD3 algorithms. 36
Figure 16: Training graph of the DDPG agent in a static environment. 38
Figure 17: Training graph of the TD3 agent in a static environment. 39
Figure 18: Training graph of the SAC agent in a static environment. 40
Figure 19: Training graph of the PPO agent in a static environment. 40
Figure 20: Multi agent benchmarking from scratch and transfer learning. 42
Figure 21: Multi agent training graph with incremented reward. 43
Figure 22: Training graph of a single agent in a changing environment. 43
Figure 23: AWS pricing calculator for the required server requirements. 44

List of Tables:

Table 1: Actor-Critic agents used in the project. 20
Table 2: Structure of all Actor-Critic agents. 32
Table 3: Neural network representation for all used agents. 37
Table 4: Benchmarking of four algorithms in a static environment. 41

1 Introduction

1.1 Motivation

The world as we know it today is in constant expansion and industrialization of using robotics is intensifying. This allows a large number of advantages in countless areas that are present in our day to day.

As examples: today specialised surgeons are capable of performing surgeries in real time remotely through highly precise robotic tools; the incredible feat of rockets that, once their mission is accomplished, land themselves back on earth; the biometrics; modulations of light lasers in fiber optic links; real-time monitoring of satellite clouds; the humanization of robots or the automotive market for autonomous vehicles.

All these systems are governed by the powerful tool of artificial intelligence. With it, the machines are capable of making their own decisions according to the adversity facing in front of them, through the type of training algorithm for which they have been destined to develop.

Thus, the main motivation of this project consists in the implementation of artificial intelligence algorithms such as DDPG, TD3, PPO or SAC, detailed in sections 2.2.2, 2.2.3, 2.2.4 and 2.2.5, which fit specifically the branch of Reinforcement Learning included within Deep Learning and Machine Learning.

These algorithms will be used to train robots, which will be in an environment with static or mobile obstacles (similar to what a warehouse would have), with the aim of reaching a destination point without having collided with any wall and all this done in an acceptable time.

1.2 Objectives

The purpose of this project is the development of reinforcement learning systems and algorithms for autonomous vehicle environments. Is both a mix from scratch work and a previous project called “Warehouse Robot Reinforcement Learning”, which is a Deep Learning course[7] from Mathworks focused on Reinforcement Learning and done entirely with Matlab and Simulink.

As it was mentioned in the previous section, the main objective of this project is to train one or more robots via AI to reach a certain goal spot through a trajectory without colliding with any static or mobile obstacle.

Thereby, to be able to do this research, since some deep reinforcement learning techniques will be used, a brief exposition of the different basic types of artificial intelligence (that will be more detailed in their corresponding sections), basic neural network concepts, and a state of the art of methods used in autonomous driving will be done.

To continue, deep reinforcement learning tools and the DDPG, TD3, PPO and SAC algorithms previously mentioned are going to be developed to see their reliability and to facilitate learning and the safety of vehicles in both static and changing environments.

Finally, and hand to hand with the above objective, an analysis of how the robot interacts and gathers information from the environment will be conducted.

This project had as initial objectives, the development of AI, Reinforcement tools for Autonomous Guided Vehicles (AGV) in changing and multi agent scenarios. For the development of this project, all have been virtual models of the scenarios and the agents have been used.

Due to the fact this work mostly uses artificial intelligence techniques and tools, firstly an introduction of the basic concepts and types of AI neural networks will be done.

1.3 Artificial Intelligence

Artificial Intelligence (AI) is the combination of algorithms with the purpose of creating machines that have the same capabilities as the human being. A technology that is still distant and mysterious to us, but that for some years now has been present in our day to day.

The artificial intelligence world can be sorted into four big categories[1]:

1. Reactive AI
2. Limited memory AI
3. Theory of mind AI
4. Self-aware AI

Reactive AI

This is the most basic type of artificial intelligence and is programmed to provide a predictable output based on the input it receives.

Reactive machines always respond to identical situations in the exact same way every time, and they are not able to learn new actions or conceive of past or future.

Different examples of reactive AI are:

- Spam filters that allow us to discern between fraudulent emails or "phishing";
- Netflix recommendation engine;
- Deep Blue, the chess-playing IBM supercomputer that bested world champion Garry Kasparov.

Limited memory AI

Limited memory AI acts by learning actions or data from the past. This is the most common type of AI used nowadays and uses observational data combined with pre-programmed information to make complex classifications and predictions.

Furthermore, this mode of technology is very used in autonomous vehicles to observe the environment such as other cars, speed, direction and information about the road.

Theory of mind AI

One of the most iconic goals in Theory of mind AI is the artificial intelligence of the robots, whose aim is to provide true decision making capabilities. For example, to understand and remember emotions and then use that information to adjust the behaviour for interacting with people.

Currently these purposes are still being heavily researched due to its huge complexity of artificial human behaviour.

Self-aware AI

The Self-aware AI is very similar to the Theory of mind AI but in this case is more focused on the awareness of the own emotions of the machines. This means that they will have a level of consciousness and intelligence similar to human beings.

They are even capable of making inferences such as "I'm feeling angry because someone cut me off in traffic".

Nevertheless, humans haven't developed this type of intelligence yet and for the moment the needed algorithms or hardware are not available.

Once the previous classes of AI were seen, some machine learning techniques will be presented below:

- Unsupervised learning
- Supervised learning
- Self-supervised learning
- Reinforcement learning

Unsupervised Learning

Unsupervised Learning[2] is a machine learning technique that doesn't require the supervision of the model by the users. Instead, the own model works by itself to discover patterns and information that was previously undetected. It uses unlabelled data.

Unsupervised learning algorithms include clustering, anomaly detection, neural networks, etc. This kind of method can allow more complexity in processing tasks compared to Supervised learning, however it normally leads to more unpredictable results.

One simple example to explain this technique is to think about a baby and a dog. First the baby sees the animal for the first time and gets used to it over time, then someone brings another different dog and the baby still can recognize the new friend as a dog, even though it's not the original one. Basically the baby wasn't taught but it learned from the data of the dog.

Supervised Learning

Supervised Learning[3] uses labelled datasets to train algorithms to classify data or predict outcomes accurately. The training datasets include correct inputs and outputs which allow the model to learn with time. The algorithm measures its accuracy through the loss function, adjusting it until the error has been sufficiently minimised.

Supervised learning can be separated in two big types of problems:

- **Classification:** This type of algorithm assigns test data into specific categories. It makes some conclusions about how the entities that are recognized within the dataset are labelled or defined. Some classification algorithms are support vector machines (SVM), linear classifiers, decision trees, k-nearest neighbour etc.
- **Regression:** It's used to understand the relationship between dependent and independent variables. It's easy to see this algorithm applied to make projections, such as for sales revenue for a given business.

Self-Supervised Learning

Self-Supervised Learning[4] learns from unlabelled data and it can be considered as a form of machine learning between Supervised and Unsupervised Learning.

This technique, as the previous ones, involves neural network working (which will be explained in the following sections). In this case the first step is to solve the task based on pseudo-labels that help to initialise the network weights. Then the own task is computed with Supervised or Unsupervised Learning.

For example, Facebook uses this method for speech recognition, which is commonly used for audio processing.

The method can be grouped into two categories which are positive and negative episodes. A positive example would be those who match the target and the negative in the opposite case, such as correctly classifying images of birds. Moreover, in this section two types of techniques can be also found:

- **Contrastive SSL:** It uses both negative and positive examples. The corresponding loss function minimises the distance between positive samples and maximises the negative ones.
- **Non-contrastive SSL:** It uses only positive examples and, instead of reaching the expected identity function with zero loss, it converges on a useful local minimum.

Reinforcement Learning

Reinforcement Learning[5] consists in how an intelligent agent learns by taking observations from an environment and making corresponding actions from that acknowledged information by taking a cumulative reward.

This kind of AI learning differs from Supervised Learning in the fact that it doesn't need labelled input/output data pairs. Alternatively, the aim is to find a balance between exploration (of uncharted territory) and exploitation (of current knowledge). A simplified scheme of how the learning works is shown below:

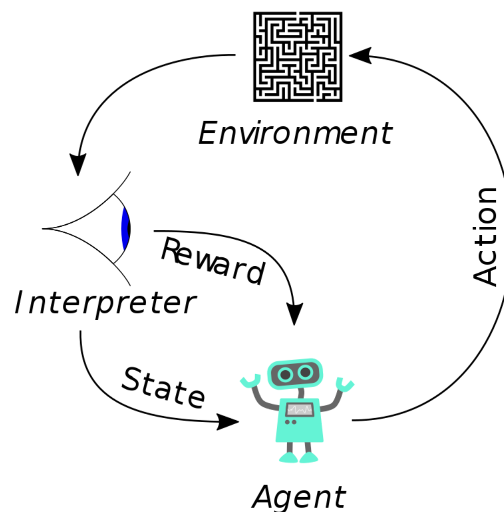


Figure 1[5]: Typical Reinforcement Learning Scenario.

As Reinforcement Learning is very widely used, it can be found in many categories such as game theory (AlphaGo[6] program, which defeated the world's best Go player in the world), control theory, multi-agent systems, statistics and also very popular in

autonomous driving, which is the computed method in the "Warehouse Robot" project above mentioned that will be later explained in detail.

1.4 Gant diagram, work packages, tasks and milestones

All this information can be seen in the Working Plan and Critical Review, so it was considered not worth repeating it here.

2 State of the art of the technology used or applied in this thesis:

2.1 Reinforcement Learning

As it was briefly mentioned in the introduction, Reinforcement Learning consists in how an intelligent agent learns by taking observations from an environment and makes corresponding actions from that acknowledged information by taking a cumulative reward.

A scheme of how the RL problems are generally modelled is shown below:

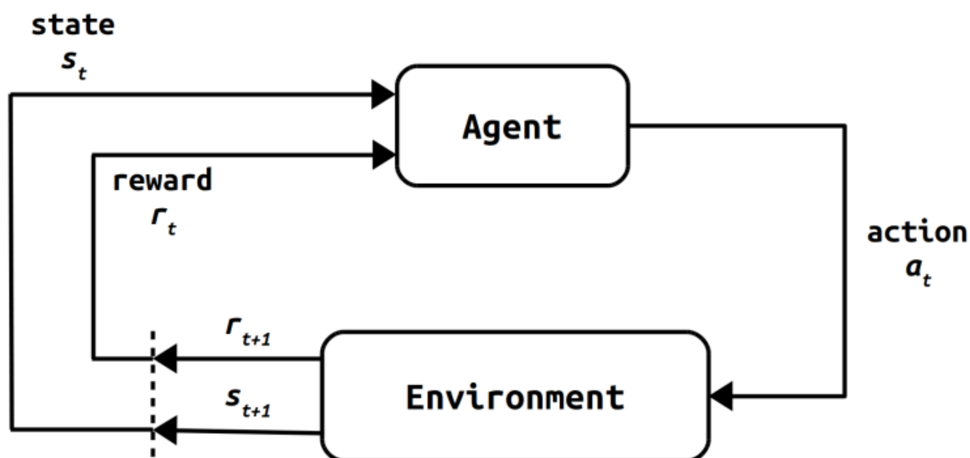


Figure 2[8]: Generalised Reinforcement Learning scheme.

The definition of the elements of a basic RL scheme are:

- State **s**: It's a representation of the environment built from a set of variables related to the problem. Therefore, the *state space* is the set of variables and all the possible values assigned with them. Thus, a state is an instantiation of a state space. Usually, the states can be called as *observations* due to the fact that the agent normally doesn't have the full knowledge about the states of the environment.
- Action **a**: The agent chooses an action from a set of actions that the environment provides at each time step. The environment is affected by the agent depending on what actions the agent makes, and for that reason is kind of a feedback process. In the case of a virtual environment, there's a function that does this mapping between states and actions towards next states, known as *transition function* or *transition probabilities*.
- Reward **r**: he environment feeds back the agent with a reward depending on the last action taken and how well it works to approach the goal. This signal is computed by the *reward function*. The objective of the agent is to maximise the overall reward it receives.
- Policy **π(s)**: Seen in a general way, a policy is a mapping of the states that define the environment and the actions to be taken in those states. Therefore, the policy basically determines the learning of the agent.
- Vale function **v(s)**: The behaviour of the value function is similar to the reward function. Both define how good a state is in the current instant but with the main

difference that in this case this value function forecasts the amount of reward that the agent can expect to accumulate in the future, starting from that state.

To sum up, the textual way of the general behaviour of the system is[9]:

The agent and environment interact at each step of a sequence of discrete time steps, t = 0, 1, 2, 3, ... At each time step *t*, the agent receives a representation of the environment's state, $S_t \in S$, where S is the set of possible states, and on that basis selects an action, $A_t \in A(S_t)$, where $A(S_t)$ is the set of actions available in state S_t . One time step later, and in part as a consequence of its action, the agent receives a numerical reward, $R_{t+1} \in R \subset \mathbb{R}$, and a new state, S_{t+1}.

At each time step, the agent implements a mapping from states to the probability of selecting each possible action. This mapping is called the agent's policy and is denoted π_t, where $\pi_t(a|s)$ is the probability of selecting action $A_t = a$ if $S_t = s$.

Also a very important concept is *discounting*. According to this approach, the agent tries to select actions so that the sum of the discounted rewards it receives over the future is maximised. To be more precise, the agent selects A_t to maximise the expected *discounted return* defined as $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$. (1)

2.1.1 Markov property

The great mathematical framework on which RL problems are based are the Markov decision processes[9][34] (MDP), but before going into detail about the mathematical concepts, the Markov property must be defined first.

For environments where it is assumed to be a finite number of states and reward values, this allows to work in terms of sums and probabilities instead of integrals and probability distributions have to be used.

Considering an environment at time *t+1* that comes from executing an action at time *t*, this answer should depend on everything that has happened before. This dynamic can be defined by specifying only the concrete probability distribution.

$$Pr\{R_{t+s} = r, S_{t+1} = s' \mid S_0, A_0, R_1, \ldots, S_{t-1}, A_{t-1}, R_t, S_t, A_t\} \quad (2)$$

For all *r*, *s'* and all possible values in previous steps: S_0, A_0, R_1, …, S_t, A_t, R_t. If the state has the *Markov property*, the response at time *t+1* depends only on the state and the action chosen in the previous step and can be defined as follows.

$$p(s', r|s, a) = Pr\{R_{t+1} = r, S_{t+1} = s' \mid S_t, A_t\} \quad (3)$$

For all *s'*, s_t and a_t. In other words, the environment satisfies the Markov properties if and only if (2) is equal to (3). In this case, the environment is said to have the Markov property.

This means that the future is independent of the past, meaning that the current state has all the information from the past states and hence, it wouldn't be relevant to keep extra information from them.

2.1.2 Markov decision processes (MDP)

Now that the Markov Property was explained in the above section, the MDP[9] can be explained. Any RL problem that meets the above condition is called a MDP and, if its state and action spaces are finite then it's known as finite MDP, which is very relevant for RL.

A finite MDP is defined by sets of actions and states on each environment step. Given any state and action *s* and *a*, the probability of each possible pair of next state and reward, *s'* and *r*, is written

$$p(s',r|s,a) = Pr\{S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a\}. \ (4)$$

As for the above equation (3), some elements of the environment such as the expected rewards for state–action pairs can be obtained,

$$r(s,a) = \mathbb{E}[R_{t+1} \mid S_t = s,\ A_t = a] = \sum_{r\in\mathbb{R}} r \sum_{s'\in S} p(s',r \mid s,a). \ (5)$$

As it was seen in section 2.1, the value functions try to predict the amount of reward that the agent can expect to accumulate in the future, starting from that state. Thereby it can be defined as:

$$V_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s]. \ (6)$$

This equation means that the value of a state s under a policy π, denoted $V_\pi(s)$, is the expected return when starting in *s* and following π.

The state-action value function can be defined as the value associated with taking an action from a state s following a policy π (6).

$$Q_\pi(s,a) = \mathbb{E}_\pi[G_t \mid S_t = s,\ A_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s,\ A_t = a]. \ (7)$$

In both cases there is the so-called discount factor represented as γ that allows to discount the value of rewards over time. That is, the necessity to tell the agent that it is better to get a positive reward sooner. Therefore, a positive real value (< 1) is commonly used so that the value of future rewards can be exponentially discounted.

Moreover, there are the optimal value functions, which are functions that try to enhance the policies for all the states notated for all the states *s* ∈ S as:

$$V_*(s) = max_\pi V_\pi(s). \ (8)$$

Optimal policies also share the same optimal action-value function, denoted Q_*, and defined for all *s* ∈ S and a ∈ A(s) as:

$$Q_*(s,a) = max_\pi Q_\pi(s,a). \ (9)$$

Finally the expressions of Q_* and V_* can be combined in this manner[33]:

$$Q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma V_*(S_{t+1}) \mid S_t = s,\ A_t = a]. \ (10)$$

2.2 Reinforcement Learning Agents

2.2.1 Actor-Critic Agents

The Actor-Critic agents are, as their name describes, agents that are built with one actor (that returns as output the action that (often) maximises the predicted discounted cumulative long-term reward) and one or more critics (that return the predicted discounted value of the cumulative long-term reward)[10].

Since the observation (states) and action spaces are both continuous in this project, the following used types of Actor-Critic agents are listed below[11]:

Agent	Type	Action Space
Deep Deterministic Policy Gradient (DDPG)	Actor-Critic	Continuous
Twin-Delayed Deep Deterministic Policy Gradient (TD3)	Actor-Critic	Continuous
Soft Actor-Critic (SAC)	Actor-Critic	Continuous
Proximal Policy Optimization (PPO)	Actor-Critic	Discrete or Continuous

Table 1: Actor-Critic agents used in the project.

2.2.2 Deep Deterministic Policy Gradient (DDPG)

The Deep Deterministic Policy Gradient[12] is an algorithm that seeks for the optimal policy that maximises the expected cumulative long-term reward.

As an Actor-Critic agent, it uses a *deterministic policy actor* π(S) and a *Q-Value function critic* Q(S,A)[13].

Whilst the agent is training:

- The actor and critic properties are updated at each time step.
- A circular experience buffer is used to store the past experiences of the agent and then, the agent updates the actor and critic by selecting a randomly sampled set of those experiences called mini-batch.
- Finally to induce some randomness, a stochastic noise is used to perturb the actions chosen by the policy at each time step.

The DDPG agent uses four function approximators to estimate its policy and value function:

- Actor **π(S;θ)**: The actor's parameters are defined as θ and its goal is to return the action that maximises the long-term reward by taking observations defined as *S*.
- Target actor **π_t(S;θ_t)**: The target actor is used to improve the stability of the algorithm. Basically, the agent periodically updates the target actor parameters θ_t from the latest actor parameters values.
- Critic **Q(S,A;ϕ)**: The critic's parameters are ϕ and returns the corresponding expectation of the long-term reward by taking observations *S* and actions *A* as inputs.

- Target critic **$Q_t(S,A;\phi_t)$**: The target critic acts exactly the same as the target actor, but with the difference that now the agent periodically updates the target critic parameters ϕ_t using the latest critic parameter values.

Both actor-target actor and critic-target critic have the same structure and parametrization. The agent updates the parameter values θ during training and, once the training is finished, those parameters remain unchanged and the trained actor function approximator is stored in π(S).

The pseudocode[12] for the training algorithm is the following one (for maintaining the notation, the μ can be seen as π, which refers to the policy) :

DDPG Algorithm

Randomly initialise critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialise target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialise replay buffer R
for episode = 1, M **do**
Initialise 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's neural network parameters θ^Q by minimising 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:

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

,where J is the start distribution of the reward return.

Update the target networks with $\tau \ll 1$:

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

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

end for
end for

2.2.3 Twin-Delayed Deep Deterministic Policy Gradient (TD3)

The TD3[14] is an extension of the DDPG algorithm. Since the DDPG agents could overestimate value functions, this post-improved version has the following modifications:

- Learns two Q-Value functions critics Q(S,A) instead of one, and uses the minimum value function estimate during policy updates.
- Updates the policy and targets less frequently than the Q functions.
- In a similar way and like the DDPG algorithm, the TD3 agent adds noise to the target action, which makes the policy less likely to exploit actions with high Q-value estimates.

Depending on the number of critics the agent has, there are two types of trainings:

- **TD3**: This algorithm includes the previous three modifications by using two Q-Value critics.
- **Delayed DDPG**: This algorithm only uses one Q-Value critic and trains a DDPG agent with target policy smoothing and delayed policy and target updates.

The behaviour of the TD3 agent is the same as the DDPG during training (section 2.2.2).

A TD3 agent utilises function approximators to update its policy and value function but with slight differences in the number of critics rather than the DDPG.

Added to the deterministic policy actor $\pi(S;\theta)$ and target actor $\pi_t(S;\theta_t)$, the new algorithm may use more than one critic:

- One or two Q-value critics $\mathbf{Q_k(S,A;\phi_k)}$: Each one with different parameters ϕ_k, returns the predicted long-term reward by taking observation *S* and actions *A* as inputs.
- One or two target critics $\mathbf{Q_{tk}(S,A;\phi_{tk})}$: The agent periodically updates the target critic parameters ϕ_{tk} with the previous ones from the current critic in order to increase its stability.

Both actor-target actor and critic-target critic (there may be more than one) have the same structure and parametrization. If there are two critics, the TD3 works better if both have the same structure but they must have different initial values.

The pseudocode of the TD3 training algorithm is detailed below:

<u>TD3 Algorithm</u>

Initialise critic networks $Q_{\theta 1}$, $Q_{\theta 2}$, and actor network π_φ with random parameters θ_1, θ_2, ϕ.
Initialize target networks $\theta'_1 \leftarrow \theta_1$, $\theta'_2 \leftarrow \theta_2$, $\phi' \leftarrow \phi$
Initialise replay buffer $\mathfrak{B}$
for t = 1 **to** T **do**

Select action with exploration noise $a \sim \pi_\phi(s) + \varepsilon$, $\varepsilon \sim \mathcal{N}(0,\sigma)$ and observe reward *r* and new state *s’*
Store transition tuple *(s, a, r, s’)* in $\mathfrak{B}$

Sample mini-batch of N transitions *(s, a, r, s’)* from $\mathfrak{B}$
$\overline{a} \leftarrow \pi_{\phi'}(s') + \varepsilon,\ \varepsilon \sim clip(\mathcal{N}(0, \overline{\sigma}), min\ action, max\ action)$, where the added noise is clipped to avoid error introduced by using values of impossible actions.
$y \leftarrow r + \gamma\, min_{i=1,2} Q_{\theta'_i}(s', \overline{a})$

Update critics $\theta_i \leftarrow argmin_{\theta_i} \frac{1}{N} \Sigma (y - Q_{\theta_i}(s, a))^2$

if *t* mod *d* **then**

Update ϕ by the deterministic policy gradient:

$$\nabla_{\phi} J(\phi) = \frac{1}{N}\sum \nabla_{a} Q_{\theta_1}(s,a)|_{a=\pi_{\phi}(s)} \nabla_{\phi}\pi_{\phi}(s)$$

Update target networks:

$$\theta'_i \leftarrow \tau\theta_i + (1-\tau)\theta'_i$$

$$\phi' \leftarrow \tau\phi + (1-\tau)\phi'$$

end if

end for

2.2.4 Proximal Policy Optimization (PPO)

The PPO[15][16][17] agent uses a type of policy gradient training that alternates between sampling data through environmental interaction, and optimising a clipped surrogate objective function using stochastic gradient descent.

The clipped surrogate objective function is used to improve the training stability by limiting the size of the policy change at each step.

The PPO is a simplified version of the TRPO[18], which is more computationally expensive than the PPO; in this case however, the TRPO agent is not used because it's not as robust as the PPO's with high dimensional observation sets.

This agent uses one value function critic *V(S)* and a stochastic policy actor π(S). During training the PPO algorithm:

- Based on the probability distribution, the agent selects random actions and estimates probabilities of taking each action in the action space.
- Before using mini-batches to update the actor and critic properties over multiple epochs, the agent interacts with the environment for multiple steps using the current policy (This is determined by the *ExperienceHorizon* parameter to be commented later on).

To estimate its value function and policy, the PPO agent has two function approximators:

- Actor **π(A|S;θ)**: The parameters of the actor are θ, and it returns the conditional probability of taking each action *A* when in state *S* as it follows:
    - Discrete action space: The probability of taking each discrete action. The sum of the probabilities across all actions is equal to one.
    - Continuous actions space: The mean and standard deviation of the Gaussian probability distribution for each continuous action.
- Critic **V(S;ϕ)**: The critic, with parameters ϕ, takes observation *S* and returns the corresponding expectation of the discounted long-term reward.

The training algorithm is the following one:

1. Initialise the actor π(A|S;θ) with random parameter values *θ*.
2. Initialise the critic V(S;ϕ) with random parameter values ϕ.
3. Generate *N* experiences by following the current policy. The experience sequence is $S_{ts}, A_{ts}, R_{ts+1}, S_{ts+1}, \ldots, S_{ts+N-1}, A_{ts+N-1}, R_{ts+N}, S_{ts+N}$.

A_t is an action taken from the state observation S_t; S_{t+1} is the next state, and R_{t+1} is the reward received for moving from S_t to S_{t+1}.

When in state S_t, the agent computes the probability of taking each action in the action space using $\pi(A|S_t;\theta)$, and randomly selects action A_t based on the probability distribution. t_s is the starting time step of the current set of N experiences.

If the experience sequence doesn't contain a terminal state, N is equal to the *ExperienceHorizon* value. Otherwise, N is less than this value and S_N is the final state.

4. For each episode step t = ts+1, ts+2, …, ts+N, compute the return and the advantage function depending on the method chosen:
   - **Finite Horizon**: Compute the return G_t, which is the sum of the reward for that step and the discounted future reward[16].

     $$G_t = \sum_{k=t}^{ts+N} (\gamma^{k-t} R_k) + b \cdot \gamma^{N-t+1} V(S_{ts+N}; \phi)$$

     Here, b is 0 if S_{ts+N} is a terminal state and 1 otherwise. Then compute the advantage function: $D_t = G_t - V(S_t; \phi)$.
   - **Generalised Advantage Estimator**: Compute the advantage function D_t, which is the discounted sum of temporal difference errors[15].

     $$D_t = \sum_{k=t}^{ts+N-1} (\gamma\lambda)^{k-t} \delta_k, \text{ where } \delta_k = R_t + b \cdot \gamma V(S_t; \phi) .$$

5. Learn from mini-batches of experiences over K epochs (cycle through the full training dataset). For each epoch:
   A. Sample a random mini-batch data set of size M from the current set of experiences.
   B. Update the critic parameters by minimising the loss L_{critic} across all sampled mini-batch data.

   $$L_{critic}(\phi) = \frac{1}{M} \sum_{i=1}^{M} (G_i - V(S_i; \phi))^2 .$$

   C. Normalise the advantage values D_i based on recent unnormalized advantage values.
   D. Update the actor parameters by minimising the actor loss function L_{actor} across all sampled mini-batch data.

   $$L_{actor}(\theta) = \frac{1}{M} \sum_{i=1}^{M} (- min(r_i(\theta) \cdot D_i, c_i(\theta) \cdot D_i) + w\mathcal{H}_i(\theta, S_i))$$

   $$r_i(\theta) = \frac{\pi(A_i|S_i;\theta)}{\pi(A_i|S_i;\theta_{old})}$$

   $$c_i(\theta) = max(min(r_i(\theta), 1 + \varepsilon), 1 - \varepsilon)$$

   The above parameters are:
   - D_i and G_i are the advantage function and return value for the ith element of the mini-batch.

- $\pi(A_i|S_i;\theta)$ is the probability of taking action A_i when in state S_i, given the updated policy parameters θ.
- $\pi(A_i|S_i;\theta_{old})$ is the probability of taking action A_i when in state S_i, given the previous policy parameters θ_{old} from before the current learning epoch.
- ε is the clip factor.
- $\mathcal{H}_i(\theta)$ is the entropy loss and *w* is the entropy loss weight factor.

2.2.5 Soft-Actor Critic (SAC)

The SAC[19] algorithm computes an optimal policy that maximises both the long-term expected reward and the entropy of the policy. A higher entropy value promotes more exploration. Maximising both the expected cumulative long term reward and the entropy balances between exploitation and exploration of the environment.

The SAC agent, like the TD3 algorithm, handles more than one Q-Value critics Q(S,A) to prevent the overestimation of the value function but with the main difference that SAC agents use a stochastic policy actor π(S) instead of a deterministic actor.

While the agent is training it:

- Updates the actor and critic properties at regular intervals during learning.
- Estimates the mean and standard deviation of a Gaussian probability distribution for the continuous action space, and then randomly selects actions based on the distribution.
- Updates an entropy weight term that balances the expected return and the entropy of the policy.
- Stores past experience using a circular experience buffer. It updates the actor and critic using a mini-batch of experiences randomly sampled from the buffer.

As it was already seen with the agent TD3, the SAC algorithm also uses the same critic $\mathbf{Q_{tk}(S,A;\phi_{tk})}$ and target critic $\mathbf{Q_k(S,A;\phi_k)}$ function approximators. Nevertheless, this new version takes advantage of a stochastic actor π**(A|S;**θ**)**.

The SAC agent generates mean and standard deviation outputs in the following way:

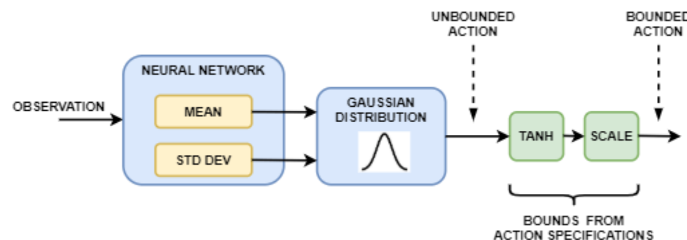


Figure 3[20]: Action selection of a SAC agent.

To select an action, the actor first randomly selects an unbounded action from a Gaussian distribution with these parameters. During training, the SAC agent uses the unbounded probability distribution to compute the entropy of the policy for the given observation.

If the action space of the SAC agent is bounded, the actor generates bounded actions by applying *tanh* and *scaling* operations to the unbounded action.

SAC agents use the following training algorithm, in which they periodically update their actor and critic models and entropy weight. Here, K = 2 is the number of critics and k is the critic index:

- Initialise each critic $Q_k(S,A;\phi_k)$ with random parameter values ϕ_k, and initialise each target critic with the same random parameter values: $\phi_{tk}=\phi_k$.
- Initialise the actor $\pi(S;\theta)$ with random parameter values θ.
- Perform a warm start by taking a sequence of actions following the initial random policy in $\pi(S)$. For each action, store the experience in the experience buffer.
- For each training time step:
    1. For the current observation S, select action a using the policy in $\pi(S;\theta)$.
    2. Execute action A. Observe the reward R and next observation S'.
    3. Store the experience (S,A,R,S') in the experience buffer.
    4. Sample a random mini-batch of M experiences (S_i,A_i,R,S'_i) from the experience buffer.
    5. Every D_C time steps, update the parameters of each critic by minimising the loss L_k across all sampled experiences:
    $$L_k = \frac{1}{M}\sum_{i=1}^{M}(y_i - Q_k(S_i, A_i, \phi_k))^2$$
    If S'_i is a terminal state, the value function target y_i is equal to the experience reward R_i. Otherwise, the value function target is the sum of R_i, the minimum discounted future reward from the critics, and the weighted entropy.
    $$y_i = R_i + \gamma * min_k(Q_{tk}(S'_i, A'_i, \phi_k)) - \alpha\, ln\, \pi(S'_i; \theta)$$
    where:
        - $\boldsymbol{A'_i}$ is the bounded action derived from the unbounded output of the actor $\pi(\boldsymbol{S'_i})$.
        - γ is the discount factor.
        - $\alpha\, ln\, \pi(S'_i; \theta)$ is the weighted policy entropy for the bounded output of the actor when in state $\boldsymbol{S}$. $\boldsymbol{\alpha}$ is the entropy loss weight.
    6. Every $\boldsymbol{D_A}$ time steps, update the actor parameters by minimising the following objective function.
    $$J_\pi = \frac{1}{M}\sum_{i=1}^{M}(- min_k(Q_{tk}(S_i, A_i, \phi_{tk})) + \alpha\, ln\, \pi(S_i; \theta))$$
    7. Every $\boldsymbol{D_A}$ time steps, also update the entropy weight by minimising the following loss function:
    $$L_\alpha = \frac{1}{M}\sum_{i=1}^{M}(- \alpha\, ln\, \pi(S_i; \theta) - \alpha\mathcal{H})$$
    , where $\mathcal{H}$ is the entropy.
    8. Every $\boldsymbol{D_T}$ steps, update the target critics depending on the target update method.
    9. Repeat steps 4 through 8 $\boldsymbol{N_G}$ times, where $\boldsymbol{N_G}$ is the number of gradient steps.
    10. Also SAC agent have target critic update methods that can be:
        - **Smoothing**: The target actor is updated at every time step with the smoothing factor τ, using $\phi_{tk} = \tau\phi_k + (1 - \tau)\phi_{tk}$.

- **Periodic**: Update the target critic parameters periodically without smoothing $\phi_{tk} = \phi_k$.

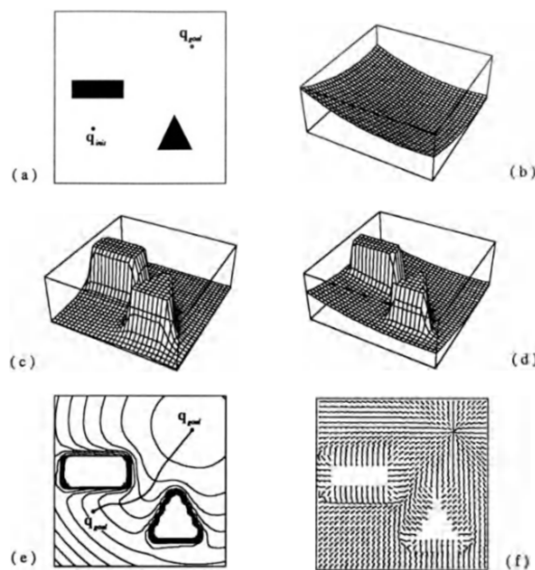
2.2.6 Potential Field Algorithm

The potential field algorithm is a technique that has been widely used in robotics for motion planning[21]. It treats the robot represented as a point in configuration space as a particle under the influence of an artificial potential field U whose local variations are expected to reflect the "structure" of the free space.

The potential function is typically (but not necessarily) defined over free space as the sum of an attractive potential pulling the robot toward the goal configuration and a repulsive potential pushing the robot away from the obstacles.

At each iteration, the artificial force $\overline{F}(q) =-\overline{\nabla}U(q)$ (11) is induced by the potential function at the current configuration, and it is regarded as the most promising direction of motion, and path generation proceeds along this direction by some increment. Potential field was originally developed as an on-line collision avoidance approach, applicable when the robot does not have a prior model of the obstacles, but senses them during motion execution[22].

This is a very important concept in AI because potential fields can be modelled as continuous mathematical reward functions (which will be used in this project). Thereby, some potential field configurations acting as obstacles are shown below:



This figure shows an attractive potential field (Figure b), a repulsive potential field (Figure c) and the sum of the two (Figure d) in a two-dimensional configuration space containing two C-obstacles (Figure a). Figure e displays both several equipotential contours of the total potential and a path generated by following the negated gradient of this function. Figure f shows a matrix of the negated gradient vector orientations over free space.

Figure 4[21]: Different potential field configurations in space between two particles.

2.3 Neural Networks

Neural networks are a wide family of machine learning algorithms that have formed the basis of the branch of Data Science and Artificial Intelligence called Deep Learning, which has obtained great results in different areas such as the classification of objects in images, behaviour prediction of users, voice recognition, etc. Artificial neural networks[26] (ANNs) are comprised of a node layers, containing an input layer, one or more hidden layers, and an output layer. Each node, or artificial neuron, connects to another and has an associated weight and threshold. If the output of any individual node is above the specified threshold value, that node is activated, sending data to the next layer of the network. Otherwise, no data is passed along to the next layer of the network (Figure 5).

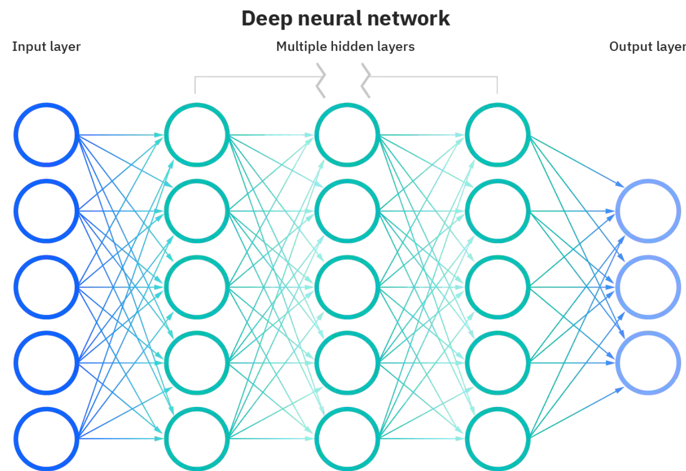


Figure 5[26]: Neural networks scheme.

The first neural network model was the "Perceptron" (Figure 6) (Rosenblatt's Perceptron 1958), which is an analogy to a biological neuron that fires an impulse if the sum of all its inputs are greater than a threshold.

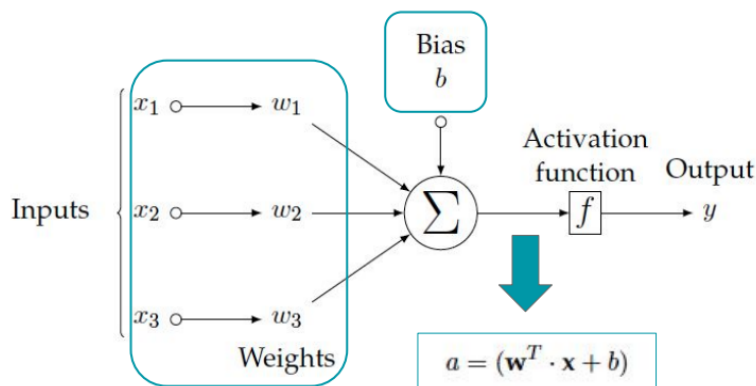


Figure 6[27]: Perceptron model.

The output y is derived from a sum of the **weighted** inputs plus a **bias** term through an **activation function**.

The activation functions introduce non-linearities in order to be able to learn more complex models than only the ones created from linear layers; it's preferable for them to

be mostly smooth, continuous and differentiable. Some common non-linear activation functions are the Sigmoid, ReLU, Softplus or tanh (which the last three ones are used in this project):

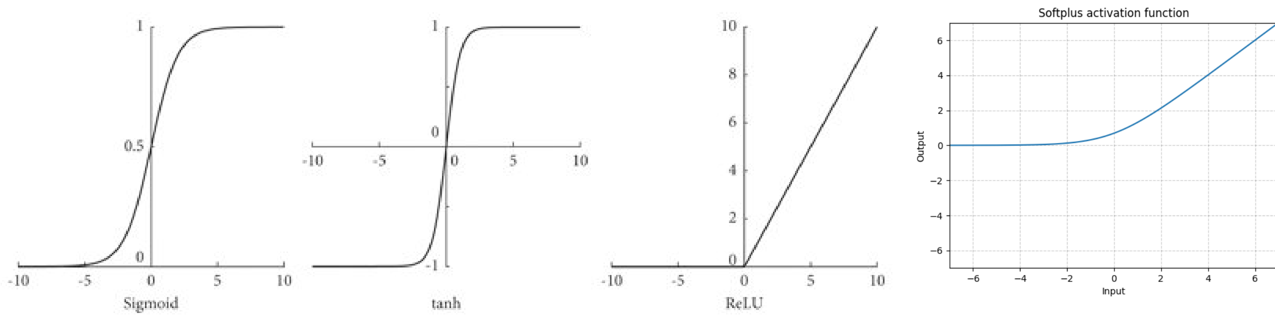


Figure 7[28]: Sigmoid, Tanh, ReLU and Softplus activation functions.

2.4 Adaptative moment estimation algorithm (Adam)

Adam[29] optimizer is the extended version of stochastic gradient descent[30] which could be implemented in various deep learning applications such as computer vision and natural language processing in the future years. The optimizer is called Adam because it uses estimations of the first and second moments of the gradient to adapt the learning rate for each weight of the neural network. Adam is a combination of two gradient descent methods: Momentum[31], and RMSP[32]:

Taking the equations used in the above two optimizers[31][32]:

$m_t = \beta_1 * m_t + (1 - \beta_1) * (\delta L/\delta w_t)$ (12) and $v_t = \beta_2 * v_t + (1 - \beta_2) * (\delta L/\delta w_t)^2$ (13); initially, both m_t and v_t are set to 0. Both tend to be more biassed towards 0 as β_1 and β_2 are equal to 1. By computing bias-corrected $\hat{m}_t$ and $\hat{v}_t$, this problem is corrected by the Adam optimizer as follows: $\hat{m}_t = m_t \div (1 - \beta_1^t)$ (14) and $\hat{v}_t = v_t \div (1 - \beta_2^t)$ (15). The gradient descents after every iteration and remains controlled and unbiased. Now substitute the new parameters instead of the old ones: $w_t = w(t - 1) - \alpha * (\hat{m}_t/\sqrt{(\hat{v}_t)} + e)$ (16).

The above parameters are:

- β, β_1, β_2: Average parameters used to control the decay rates of the moving averages.
- w_t: Weights at a time “t”.
- δL: Derivative of the loss function.
- e: Constant.
- α: Learning rate.
- m_t: Gradient (estimate of the mean).
- v_t: Square of the gradient (estimate of the uncentered variance).

3 Methodology / project development:

The aim of this project is to map the trajectories of one or more robots through an environment with obstacles with the objective of crossing the two lower obstacles through the centre. This work has been developed based on an already created project called "Warehouse Robot", which belongs to the course "Reinforcement Learning Onramp[7]" of the Mathworks website, made with Simulink and Reinforcement Learning Toolbox, that it will be called as “Static Environment” in next sections. From this previous model, dynamic and multi agent scenarios have been developed in this project.

3.1 Static Environment

First of all, the **environment** in which the agents (also called robots) are going to interact has to be defined. In the following images the scenario, observations and forces used by the agents are depicted:

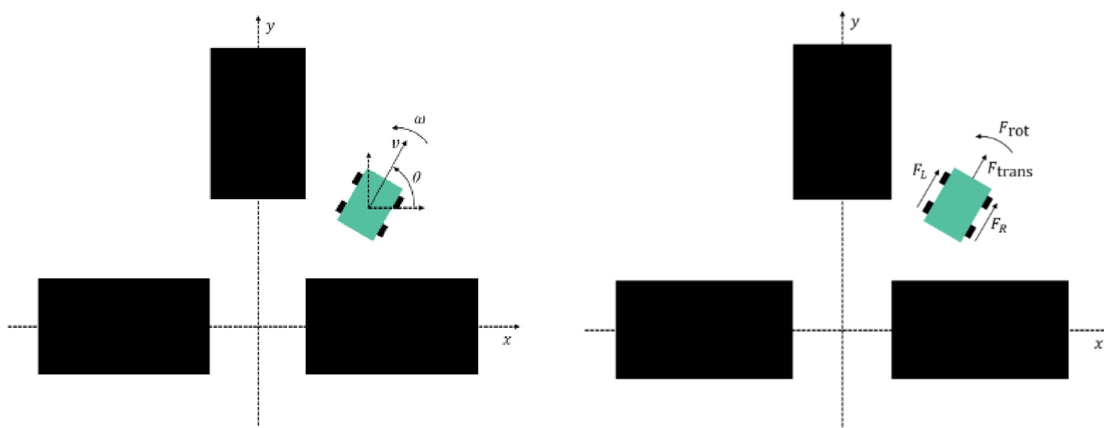


Figure 8[7]: Static environment and robot forces representation.

In this image, the three large black blocks represent the obstacles to be avoided and the robot is represented by a green rectangle (wheels are for a better understanding of the picture).

For the agent to be able to perform an action, the environment has to provide a series of **observations** in the form of six continuous variables:

- **x** and **y** agent positions: The robot position is given in terms of coordinates aligned with the shelves, so that the origin is between the three shelves.
- **sin(θ)** and **cos(θ)**: Instead of just θ, the robot uses sin(θ) and cos(θ) to orient itself. This means that the values stay in the [-1,1] range and do not jump (from 359° to 0°, for example).
- **v** and **ω**: These parameters represent the velocity and the angular velocity which allows the robot to go forwards or backwards and to spin in any direction.

Once the observations are received, the agent makes the decision to carry out **actions**, which can be split down into two rotational and translational forces (both normalised to the [-1,1] range) as follows:

- $\mathbf{F_L}$ and $\mathbf{F_R}$: Forces applied to the left and right wheels, respectively.
- **Translational force**: It's computed as $F_{trans} = \frac{1}{2} * (F_R + F_L)$ (17) and allows the robot to move forwards and backwards.
- **Rotational force**: It's computed as $F_{rot} = \frac{1}{2} * (F_R - F_L)$ (18) and enables the robot to make turns.

3.1.1 Single agent

The Simulink model for one agent and the static environment is the following one:

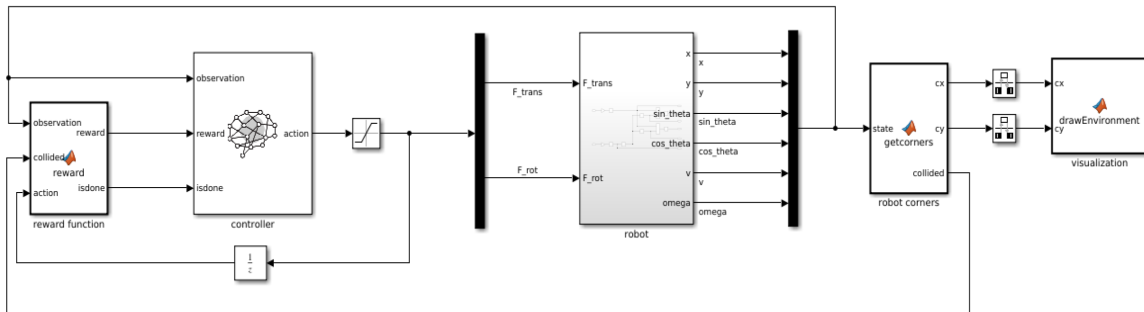


Figure 9[7]: Initial Simulink model for one agent.

The agent, "controller" block, and observations blocks and how they interact with each other are zoomed below:

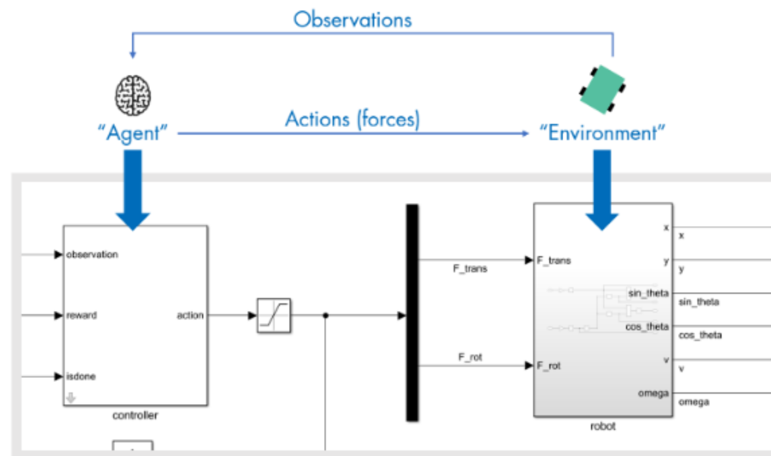


Figure 10[7]: Agent and Environment blocks of the initial Simulink model for one agent.

The entire system works as a system with feedback and, to explain its operation, we must first focus on the main blocks corresponding to the agent and the environment, the act of which was previously explained in the above section (Figure 9). That is, the environment generates a vector of six observations that are sent to the agent's block so that it decides to perform an action. This action is scaled in [-1,1] range by means of an intermediate block, and it's divided into translational and rotational forces, which are sent back to the environment, the "robot" block, for the subsequent generation of observations.

The vector of observations is connected to the "getCorners" function that obtains the central x and y coordinates of the robot, which in turn are used by the "drawEnvironment" function to visualise the interaction of the agent and the environment graphically. It also calculates if there has been some type of collision in the form of a boolean signal called "collided".

Finally, to close the cycle, the "collided" signal, the observations and the action taken by the agent, are transferred to the "reward" block so that it calculates the agent's reward and checks if the simulation is finished by the "isdone" signal. The simulation is considered a success if the robot (that appears in a random position to the right or left of the environment) manages to reach the gap between both lower obstacles for a $y<1.5$ value (Figure 12).

The DDPG, TD3, SAC and PPO algorithms were tested using the Matlab Toolbox Reinforcement Learning in this static environment to see which of them had a better performance, and later on to increase the complexity of the scenario.

The process for the implementation of the four algorithms was as follows:

For all of them, first the environment was modelled as described in section 3.1; then, the same basis was used for the structure of the neural network (seen in more detail in section 3.4) with some differences between some algorithms. Subsequently, agents were built to ultimately train and simulate them. The structure of the agents as it was programmed with Matlab is shown in the following table:

Agents	DDPG	TD3	SAC	PPO
Actors	rlDeterministicActorRepresentation	rlDeterministicActorRepresentation	rlContinuousGaussianActor	rlContinuousGaussianActor
	Quantity: 1	Quantity: 1	Quantity: 1	Quantity: 1
Critics	rlQValueRepresentation	rlQValueRepresentation	rlQValueFunction	rlValueFunction
	Quantity: 1	Quantity: 2	Quantity: 2	Quantity: 1

Table 2: Structure of all Actor-Critic agents.

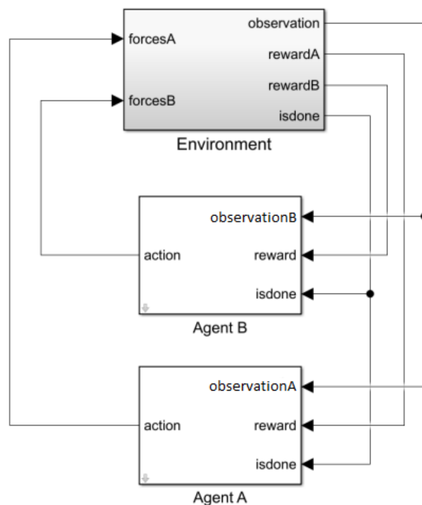
Results of the different agents are shown in section 4. The description of the methodology for the evolved scenarios is continued below.

3.1.2 Two agents

Based on the selected algorithm for a static environment, in this scenario two DDPG agents are used instead of one.

For this, the following considerations are taken:

- The sketch of the Simulink model is as follows:



The environment is enlarged so that it now provides the custom observations for each agent based on each agent's actions as input, as well as both rewards and the "isdone" signal that checks if the simulation has finished.

The functions "getCorners" and "drawEnvironment" (seen in Simulink model of section 3.1.1) were also modified so that the possibility of individual or mutual collision of the robots and the graphical representation of the simulation are added.

Figure 11: Sketch of the Simulink model for two agents.

- The simulation is considered successful if both robots manage to pass between the two lower obstacles, but now the problem is that if one of them reaches a y<1.5 value it cannot be stopped there, since, if the other agent also manages to reach it, both would collide. The proposed solution is to leave a space wide enough so that, if an agent passes through this space, it is considered that it has reached the goal and continues going down until it stops with enough distance so that, when the second robot fulfils the condition of y<1.5, it does not collide. Therefore, both robots would have achieved the objective.

3.2 Changing environment

3.2.1 Single agent

In this scenario, the novelty is that now the agent interacts with a changing environment, which adds more randomness to it.

The Simulink model shown in section 3.1.1 has now been modified so that, during the simulation, as the robot moves to try to reach the goal, the upper obstacle also appears in a random vertical position and makes a lateral movement towards one of the sides and, in case the object reaches one end of the environment and the simulation is still active, it reappears on the opposite side to continue with the same trajectory.

3.3 Providing continuous rewards

3.3.1 Static environment with a single agent

Since the set of actions and observations is not finite, the initial reward is calculated from a two-dimensional function (x,y) (Figure 12). This includes two decreasing exponentials for both axes to encourage the robot to move towards the space between the shelves ((x,y)=(0,0)) and then down through the gap (y<0). Furthermore, it penalises the robot for actions, angular velocity, collisions (boolean), and it increases the reward when there is a success (boolean) with a relevant positive reward.

The function is as follows:

$reward = 0.06exp(-3y) + 0.05exp(-8x^2) - 0.14 + penalties$ (19),

where the penalties are:

$penalties = -0.001\omega^2 - 0.01|action|^2 + 5madeit - 2collided.$

The coefficients of the exponentials and the subtraction term are selected in order to have a grid of negative values except the goal, which has a value very close to zero. This is very important because otherwise the agent could learn to spin all the time without reaching the goal if it finds itself capable of accumulating positive rewards in any other place without colliding.

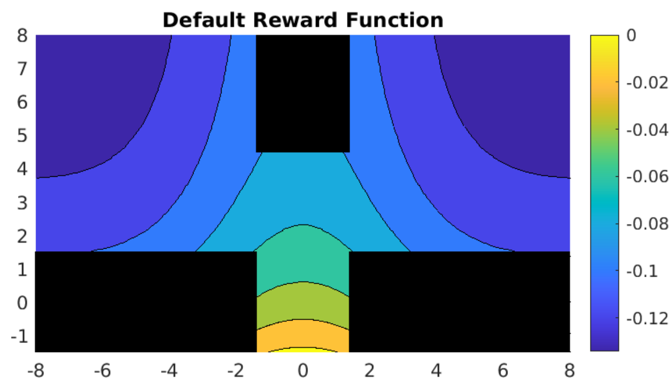


Figure 12: Default reward function.

This reward function was utilised for all the proposed algorithms and is used as a basis for the later more complex modifications of the environment.

3.3.2 Static environment with two agents

In the static environment with two agents, the original reward function (Figure 12) was modified so that now, apart from the decaying exponentials and penalties, it has overlays of various contour lines around the obstacles and the robots themselves, which are updated in real time with the movement of agents (this is another implementation of the potential fields described in section 2.2.6). This is done with the aim that the agents avoid getting too close to the obstacles and also to each other so as not to collide, since they would obtain a very negative value.

The new reward feature is visualised below:

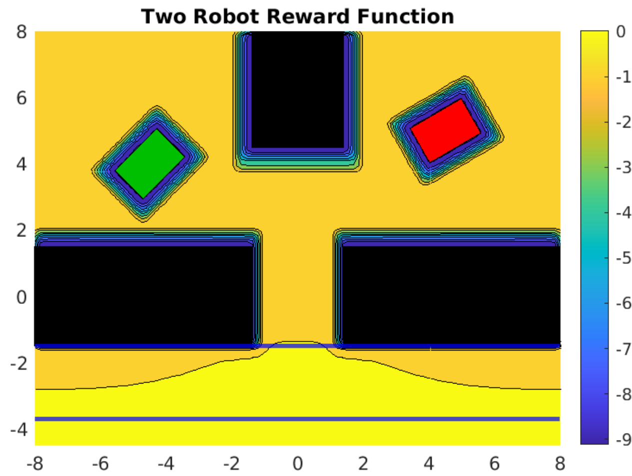


Figure 13: Two robot reward function.

Although the range of colours does not allow decreasing exponentials to be displayed as in the previous case, the desired condition of negative values is still maintained. This occurs because the contour lines are influencing the display. Also the two horizontal yellow lines are the success references for both robots, which was explained in section 3.1.2.

Another important modification is that, for this environment, the reward is not evaluated at the centre of the robot, as in previous static single agent environment. In order for the robot to be aware of the proximity of an obstacle or another robot, the new reward is evaluated as the average value of the regard at the four corners of the robot. In this way, a potential collision at one of the four corners is quickly acknowledged by the reward and the agent can react to this situation.

3.3.3 Changing environment with a single agent

For the changing environment with a single agent, the default reward function (Figure 12) was used as the base, to later add contour lines around the obstacles, which follow the upper object in real time as it moves. The graphic representation is:

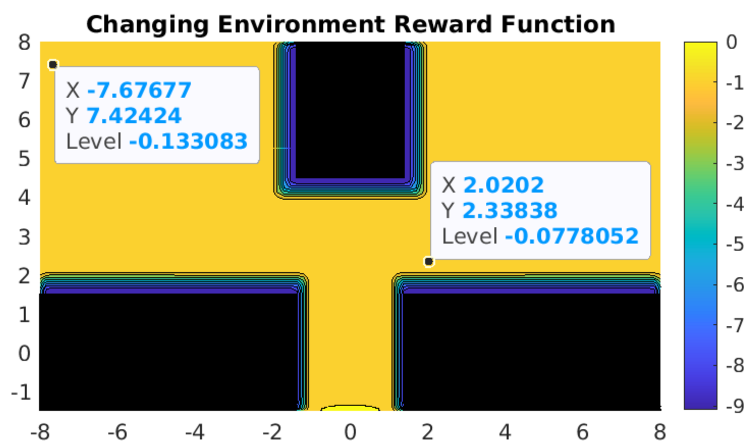


Figure 14: Changing environment reward function for one robot.

Like the reward function for two robots (Figure 13), the level curves make it difficult to graphically see the decay of values towards the target. For this reason, two markers have been placed to facilitate visualisation.

3.4 Neural networks and training

Neural networks

As previously mentioned in section 3.1.1, the DDPG agent neural network (also used for the TD3 agent but with two critics) already provided by the "Reinforcement Learning Onramp" course has been used as a base to build the actors and critics.

Both structures are as follows:

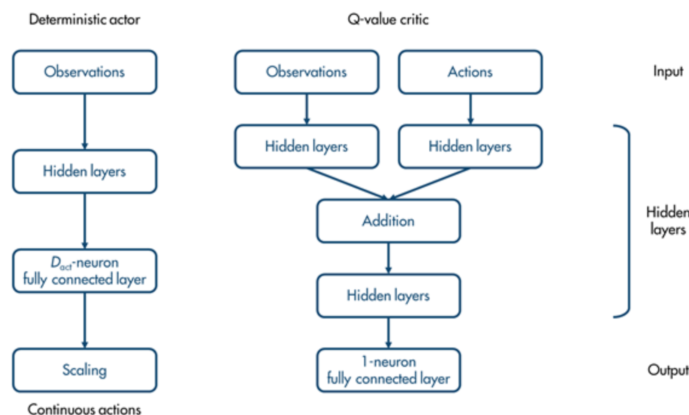


Figure 15[7]: Actor and critic neural networks for DDPG and TD3 algorithms.

As it can be seen in table 3, the structures of networks among the agents are quite similar:

- The “obs” layers are *feature input layers* for 6 inputs (observations dimension).
- The “act” layers of the actors are *fully connected layers* for 2 inputs (actions dimensions) whereas the “act” layers of the critics are *feature input layers* for 2 inputs (actions dimension).
- All the layers which contain a “fc” in their names are *fully connected layers* with 100 neurons, except for the “mp_fc2” and “vp_fc2”, which have 2 neurons (actions dimension). Moreover, the final layer named “value” of the network of critics, is also a *fully connected layer* with 1 neuron to provide a single output value.
- The “relu”, “scact” (equal to “tanh”) and “vp_out” layers, are the activation functions *ReLU, tanh* and *SoftPlus*, respectively. The “mp_out” layer does an scaling for the actions of 1.

The DDPG algorithm contains the same neural network structure for the actor and the critic of the TD3 agent, whereas it only shares it for the critic in the case of the SAC algorithm.

In PPO and SAC algorithms the actor’s network must provide two branches to compute the mean and standard deviation of the actions. Instead of the DDPG and TD3 agents, which directly outputs the action’s values.

All critics of agents except the PPO (that has only one path), use two branches with the observations and actions as inputs which are merged to compute the observations.

Agents	Actor Neural Network	Critic Neural Network
DDPG & TD3	obs fc1 relu1 fc2 relu2 fc3 relu3 act scact	obs fc1 relu1 fc2 act fcact add relu2 fc3 relu3 value
PPO	obs fc1 relu1 fc2 relu2 fc3 relu3 act scact mp_fc1 mp_relu mp_fc2 tanh mp_out vp_fc1 vp_relu vp_fc2 vp_out	obs fc1 relu1 fc2 relu2 fc3 relu3 value
SAC	obs fc1 relu1 fc2 relu2 fc3 relu3 act scact mp_fc1 mp_relu mp_fc2 vp_fc1 vp_relu vp_fc2 vp_out	obs fc1 relu1 fc2 act fcact add relu2 fc3 relu3 value

Table 3: Neural network representation for all used agents.

Training

Agent training is made up of episodes, which end if:

- There is any type of collision (either with the environment or between robots).
- The agent takes 200 steps during the episode (value chosen to ensure reasonably fast trajectories) or all the robots reach the finish line.
- The agents end the training prematurely if during 50 episodes obtains an average reward greater than or equal to 2.

4 Results

In this part, the results of several simulations carried out by the DDPG, TD3, SAC and PPO algorithms will be presented through the different scenarios seen in section 3.

First, the four algorithms are tested in the basic static, single agent environment to check which one is the most efficient, to later carry out more complex simulations.

4.1 Static Environment

4.1.1 Single Agent

All agents used in the static environment had common hyperparameters such as:

- Discount factor (γ) very close to 1.
- Maximum number of steps per simulation of 200.
- "Adam" training algorithm
- Stop training value of 2 for an average reward for 50 episodes.
- Maximum number of episodes per simulation of 100000.
- Mini-batch of 128 random samples from the experience buffer.
- For each of the used training agents: TD3, PPO and SAC; the recommended parameters by Mathworks[23][24][25] were used.

All the rewards were computed from the reward function (Figure 12) seen in section 3.1.1.

DDPG

The following image (Figure 16) shows the training graph corresponding to the DDPG agent:

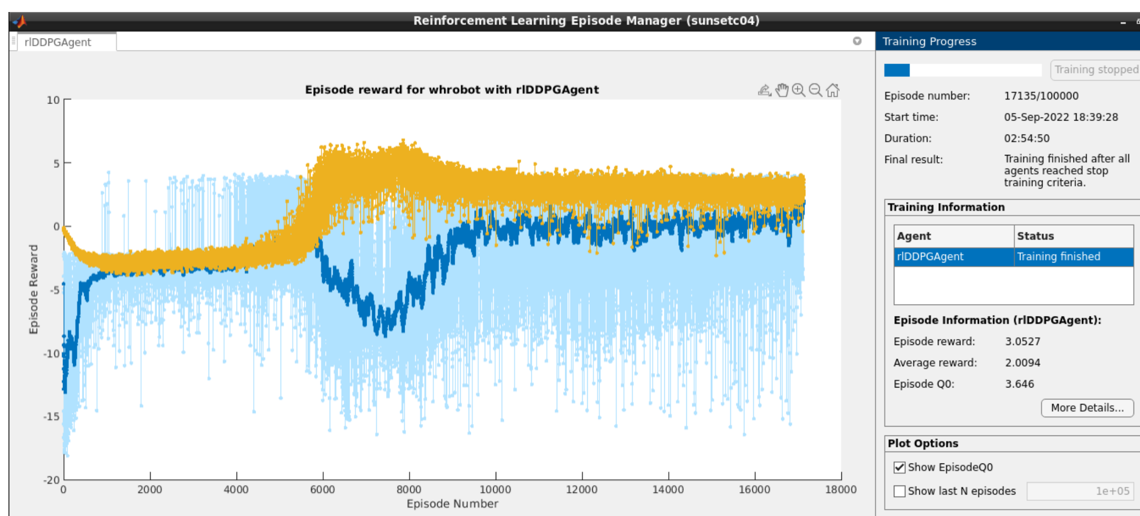


Figure 16: Training graph of the DDPG agent in a static environment.

As can be identified at the beginning of the training, the agent obtains very negative reward values (light blue curve) since he is still exploring the terrain. In turn, the estimated value of reward accumulated by the critic (yellow curve) begins to decrease and stabilise with the average reward (dark blue curve) as the robot begins to have successful

episodes. Before finishing the training due to having reached the stop training value condition at the 17135 steps, a distance appears between the average reward and the estimate of the reward by the critic, which begins to be positive. This indicates a period in which the agent accumulates very negative rewards until it stabilises again to finish the training with an average reward of 2.0094.

TD3

The visualisation of the TD3 agent training is shown below:

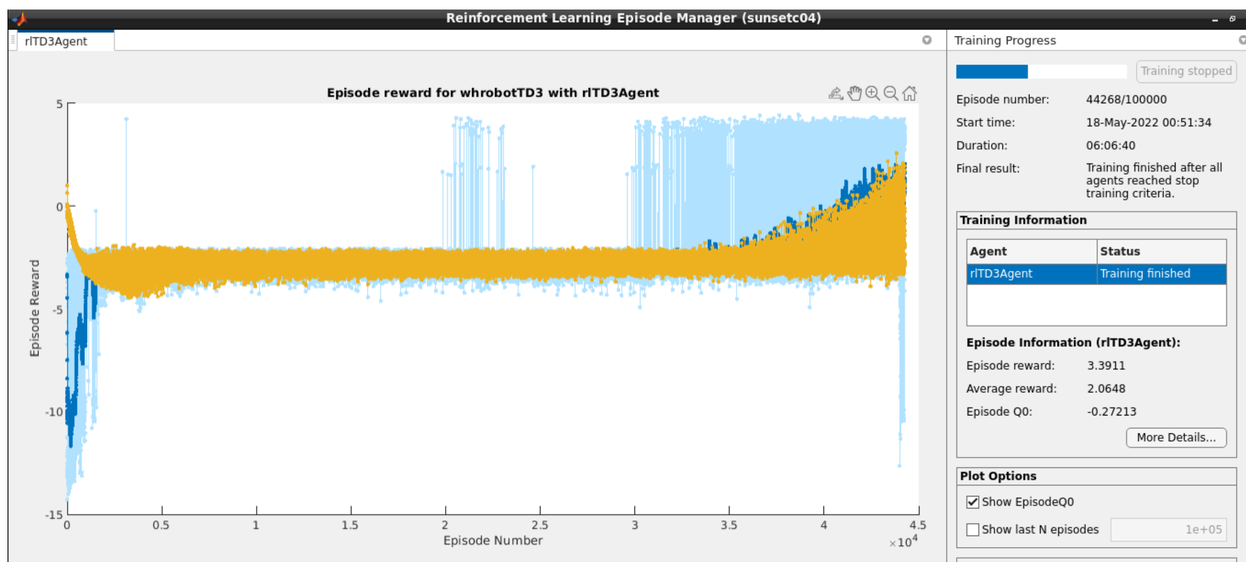


Figure 17: Training graph of the TD3 agent in a static environment.

In the same way as in the DDPG agent training graph (Figure 16), at the beginning both the agent and the critic obtain very negative reward values, which once stabilised, are maintained with few fluctuations while the agent does not yet have a solid foundation of learning. It is not until 30000 episodes, when the agent begins to obtain successful simulations more frequently (in turn, the critic begins to estimate higher future rewards) until the stop training value is obtained at around 44000 episodes with an average reward of 2.0648.

SAC

The training graph (Figure 18) of the SAC algorithm has a similar shape to the TD3 agent (Figure 17). At first, like all agents, it gets very low rewards, but with the difference that the critic creates a small peak at the beginning with positive future reward values, which, together with the average reward, stabilises as the agent learns. The value fluctuations are significantly lower than the DDPG and TD3 algorithms and, approximately over 7600 episodes, the agent ends the training complying with the stop training value and with an average reward of 2.047.

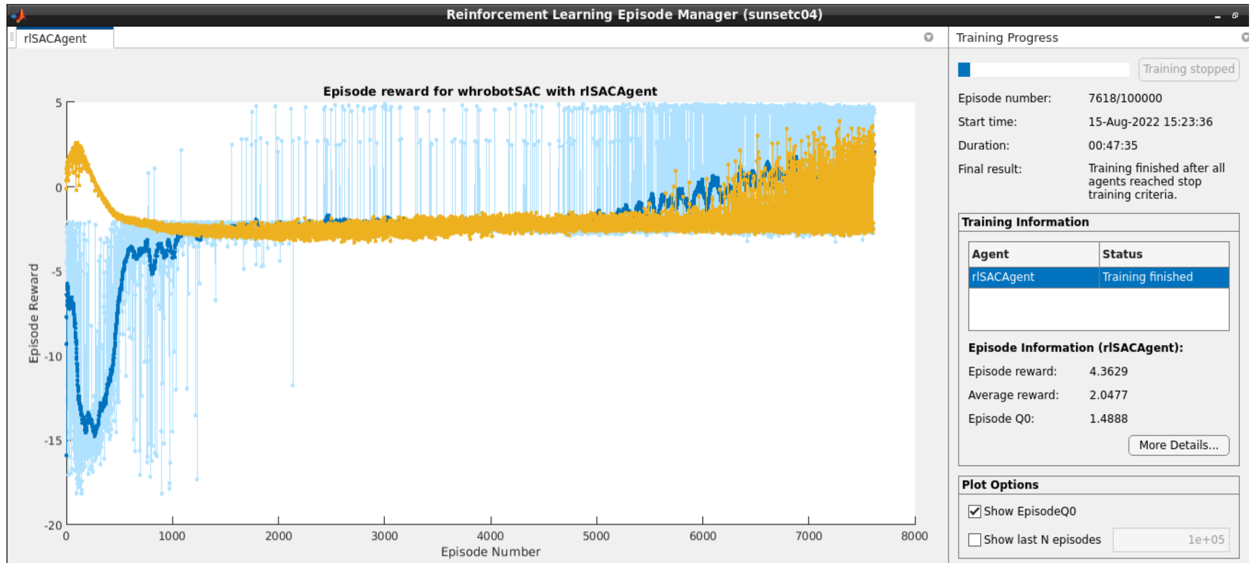


Figure 18: Training graph of the SAC agent in a static environment.

<u>PPO</u>

The training graph (Figure 19) of the PPO agent differs greatly from the rest of the algorithms. The behaviour of the agent has been quite random throughout the execution, since many fluctuations are observed in the reward and critical curves, which reflect that the agent has not been able to meet the stop training value. Therefore, the agent has finished the simulation having reached the maximum number of 10000 episodes with an average reward of practically 0.

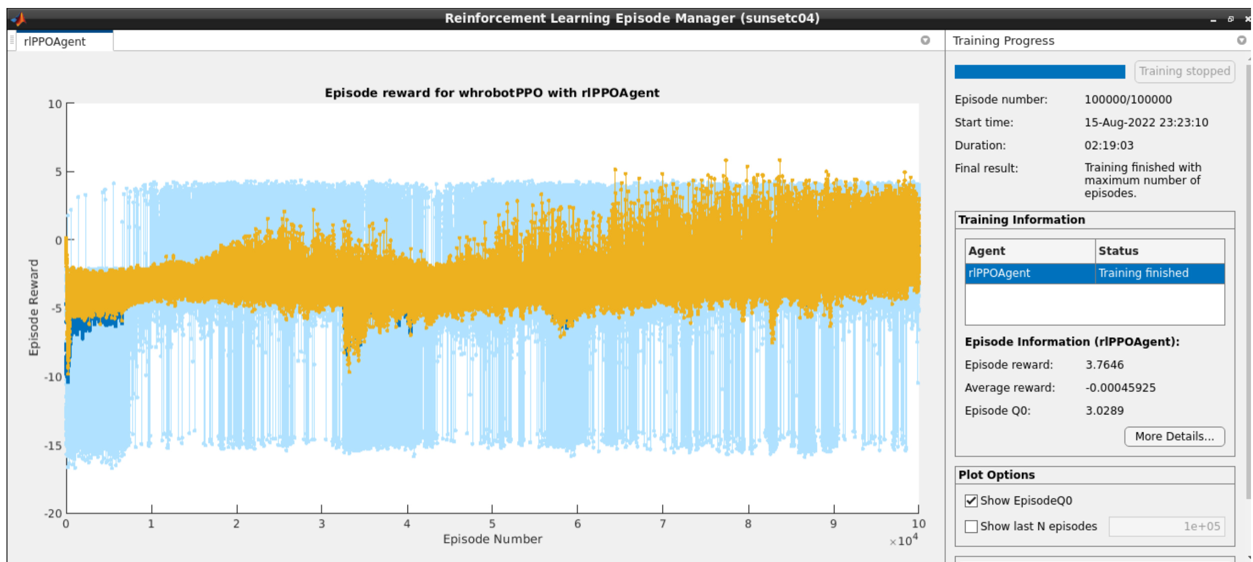


Figure 19: Training graph of the PPO agent in a static environment.

The *ExperienceHorizon* parameter was selected with a value of 512. In the first experiment it had a value of 128 and, despite the fact that the agent reached the goal in some episodes during training, it was not able to train correctly (*ExperimentHorizon* parameter should be greater or equal than the mini-batch size).

Below is shown a representative table with the benchmarking of the four algorithms for a total of 200 simulations:

Benchmarking	DDPG	TD3	SAC	PPO
Number of episodes	17135	44268	7618	100000
Average reward	2.0094	2.0648	2.0477	-0.0005
Successful runs	197	147	104	102
Collisions	3	42	96	98
No result	0	11	0	0
Mean time (s)	6.9277	7.2313	7.0577	7.3897
Reliability	99%	73.5%	52%	51%

Table 4: Benchmarking of four algorithms in a static environment.

Even though all algorithms have a similar average time and reward of approximately 7 seconds and 2 (except for the PPO agent, which has a value of practically 0), respectively, and based on the percentage of successful simulations, the DDPG agent is the most optimal one with a value of 99% and, in addition to having the lowest average time (6.9277 seconds), the number of training episodes is not disproportionately large compared to the TD3 or PPO algorithms. Although the algorithm that required the fewest episodes to finish the simulation was the SAC, its success rate is 52%, so, together with the TD3 and PPO agents, with percentages of 73.5% and 51% respectively, are discarded for the other experiments with more complex environments.

For all these reasons the best option is the DDPG agent, followed by the TD3, then the SAC and finally the PPO.

4.1.2 Two Agents

As it was shown in table 4, the most optimal agent for the single agent scenario is the DDPG. This agent is the selected one for the next experiment consisting of a static environment with two DDPG agents, whose Simulink model, reward function and parameters, were detailed in sections 3.1.2, 3.3.2 and 4.1.1, respectively.

Two experiments were carried out, in which the reward function was modified to add an extra reward of 5 or 20 upon successful completion of the simulation. This was done in order for the AI to recognize the simulation as successful, since even after reaching the destination, the total accumulated reward could become negative due to the complexity of the environment.

In addition, in the first experiment, the agents were trained from scratch, performing transfer learning every 5000 episodes for a total of 2500000. In the second one, the same pretrained agent with 100000 episodes (Figure 22) in the moving environment was used for both robots, which also performed transfer learning in steps of 1000 episodes.

The training graphs are shown below:

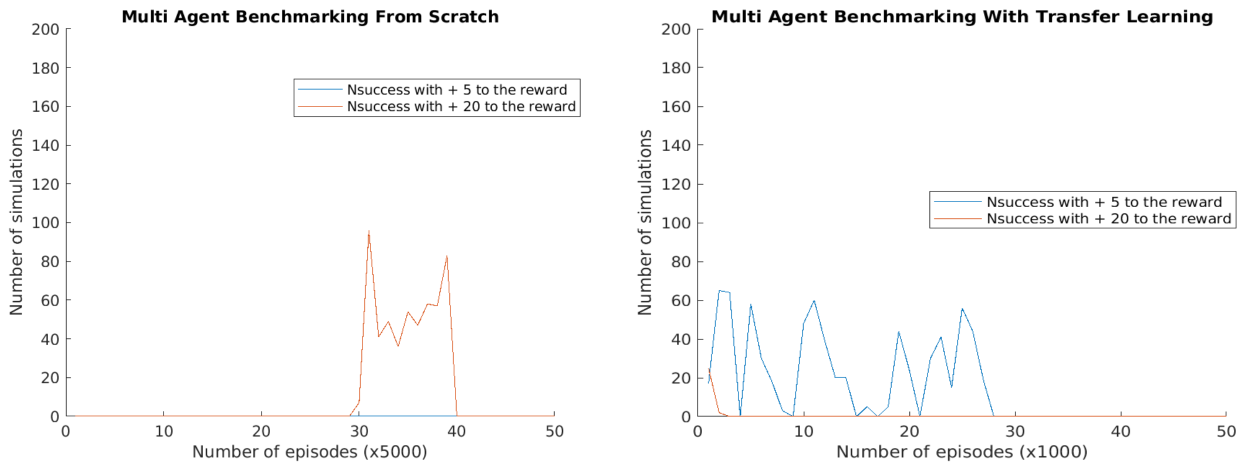


Figure 20: Multi agent benchmarking from scratch and transfer learning.

In the graph of the multi agent benchmarking from scratch, it is observed that both agents begin to obtain successful simulations around 140000 episodes when the extra reward is +20 (red curve), thus forming a peaked region limited by two maximums of 96 and 83 successes respectively. Then, it abruptly decays towards the value 0. In contrast, when the extra reward was +5 (blue curve), the agents were not able to learn to reach the goal.

In the graph of the multi-agent benchmarking with transfer learning, whose pretrained robot used the extra reward of +5 by default in its reward function (equation 19), it is observed that for this same reward, the agents already begin to obtain practically successful simulations at the beginning of training. These data fluctuate in the form of a series of peaks with maximums around 60 successes and intervals of null values, which become definitive around 27000 episodes. Instead, the red curve corresponding to the extra reward of +20 shows a total of approximately 25 successes at the beginning, which start to decay to the value 0 and remain constant.

In conclusion, agents are capable of learning, but this learning fluctuates during a certain interval of episodes and, if the algorithm is overtrained, the agents begin to act incorrectly and stop having successful episodes. Therefore, an agent belonging to that useful interval would have to be saved. Furthermore, as it is a complex environment with more than one agent, a certain randomness in the behaviour of the robots can happen.

For example in the next training graph (Figure 21) with an increased reward to have more insights, it can be observed that the second robot (bottom chart) starts prioritising the other agent's success over its own at around 37000 episodes, thus practically is no longer getting any rewards.

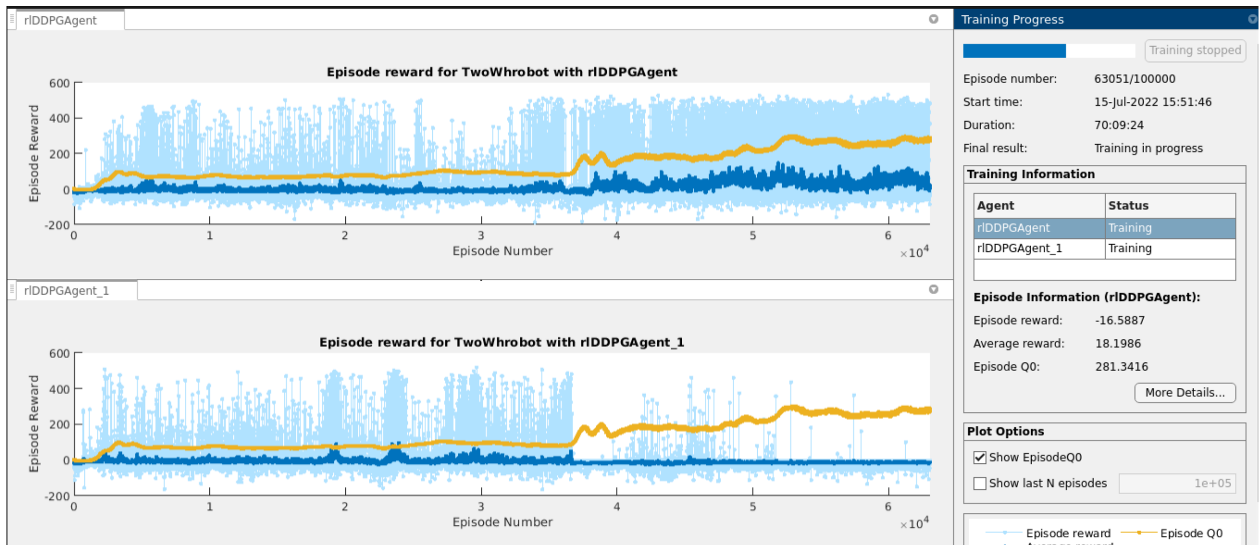


Figure 21: Multi agent training graph with incremented reward.

4.2 Changing Environment

4.2.1 Single agent

In this experiment a single DDPG agent is trained in a changing environment where the upper obstacle is also moving. The reward function seen in section 3.3 was used with contour lines updated in real time around the objects. In addition, the reward was calculated by averaging the value of the four corners of the robot so that it is able to detect the proximity of the obstacles. The training graph is shown below:

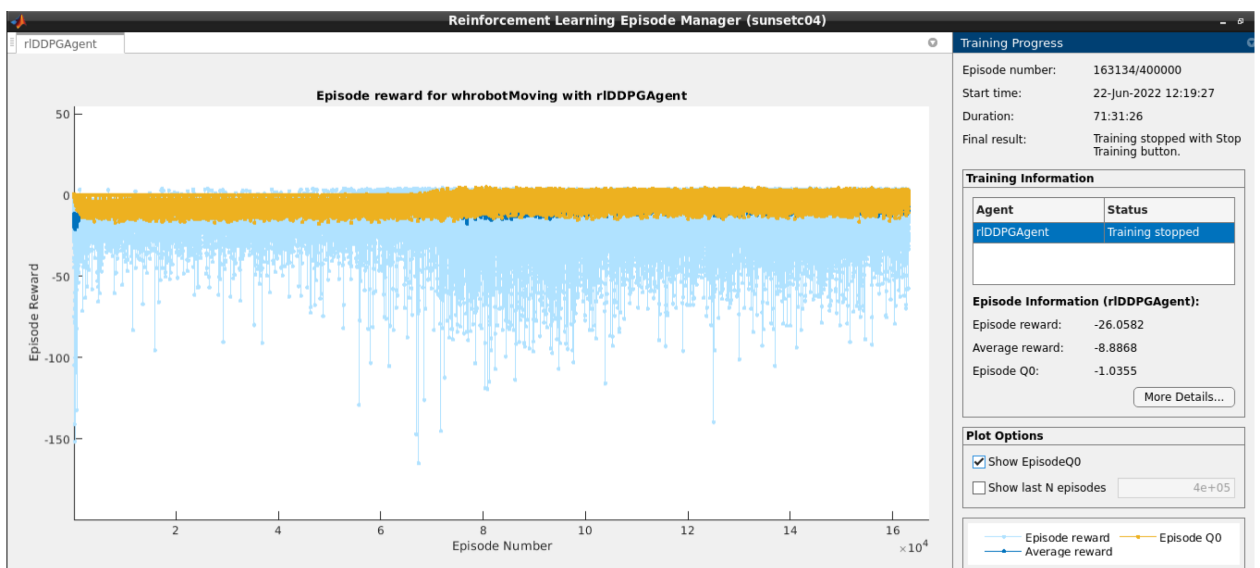


Figure 22: Training graph of a single agent in a changing environment.

The graph has a similar shape to the static environment (Figure 16) but with a much flatter characteristic peak (around 80000 episodes) between the reward and critic. This is because the average reward obtained (-8.8868) differs from the static case (2.0094), since there are now contour lines and the corners are averaged, so the simulation had to be stopped without reaching the stop training value of 2. Nevertheless, the robot has

made 184 successes and 16 collisions in 200 simulations with an average time of 7.9226 seconds. Furthermore the agent was able to do its trajectories avoiding the moving object and maintaining a safe distance with the obstacles.

5 Budget

This project does not have any physical cost, on the contrary, the cost is given at the computational and software level.

The SUNSET server of CALCULA Computing Services of the TSC Dept. (Signal Theory and Communications Dept.) has been used, as well as the student MATLAB licence. In said server, up to 20 CPUs and 128 GB of RAM were available, although in no case was it necessary to use all the resources.

Thus, it's going to proceed to analyse the monthly costs of an online server with sufficient computing capacity and the hours of work dedicated as an intern (9€/h).

The selected instance of an equivalent AWS EC2 server as the resources used at SUNSET-CALCULA server has the following details:

r6g.2xlarge

On-Demand hourly cost	vCPUs	GPUs
0.472	8	NA
1YR Std reserved hourly cost	**Memory (GiB)**	**Network performance**
0.2982	64 GiB	Up to 10 Gigabit

Figure 23: AWS pricing calculator for the required server requirements.

The price of the server is 227.17€ per month and considering that 15 hours per week as an intern were dedicated, the total budget per month is **767.17€**.

6 Conclusions and future development

Conclusions

The development of this project did not turn out to be linear at all. At first, it was quite difficult to understand and implement the algorithms, but once the first results were obtained, the path became much faster and more automatic. Matlab's Reinforcement Learning toolbox has made it possible to carry out this project in a fairly visual and intuitive way, since there are currently countless examples, online courses and documentation to consult on the internet which have accelerated my learning curve on the subject of AI.

Throughout this project, the DDPG, TD3, SAC and PPO algorithms have been implemented and tested under static and changing scenarios.

The DDPG algorithm was the one that obtained the best results in the basic static, single agent environment, so it continued to be used in the rest of the experiments.

The agent was able to reach the goal in the changing environment, detecting the proximity of objects and maintaining a safe distance by averaging the reward with its four corners. However, depending on the speed of the object and the reward function, the agent may reach the goal without being aware that an object is approaching, leading to false successes.

Within the multi agent scenario, it was shown that both robots were capable of learning satisfactorily, but that if the algorithm was overtrained, they began to make errors until they became unusable. In addition, given the new complexity of the environment, each training could yield a different result than the previous one, such as prioritising one robot over another (Figure 21).

It should be noted that it would have been practically impossible to carry out the simulations if the university's SUNSET-CALCULA server had not been available, since very high computing requirements were needed in order not to eternalize the executions of the programs and at the same time to be able to launch several threads simultaneously.

Thus, in terms of results and concepts, this project has been successfully developed; which also leads to a deeper understanding of the Deep Learning tools.

Future development

Artificial intelligence is such a complex topic that, within this same project, many experiments could have been carried out, such as, for example, what criteria to use when choosing the stop training value according to the agent and environment; the combinations of hyperparameters that best allow agents to train, or which is the most optimal neural network for a specific scenario.

Also in the future, when Mathworks implements the experiment manager for reinforcement learning (currently it is available for problems related to image classification), new research can be carried out in a much more powerful, efficient and visual way.

Bibliography:

1 Bernard Marr. “Understanding the 4 types of Artificial Intelligence (AI)”. Linkedin, 2012. Available: https://www.linkedin.com/pulse/understanding-4-types-artificial-intelligence-ai-bernard-marr

2 Daniel Johnson. “Unsupervised Machine Learning: Algorithms, type and examples”. Guru99, 2022. Available: https://www.guru99.com/unsupervised-machine-learning.html

3 IBM Cloud Education. “Supervised learning. Learn how supervised learning works and how it can be used to build highly accurate machine learning models”. 2020. Available: https://www.ibm.com/cloud/learn/supervised-learning

4 Bouchard, Louis. "What is Self-Supervised Learning? Will machines ever be able to learn like humans?". 2020. Available: https://medium.com/what-is-artificial-intelligence/what-is-self-supervised-learning-will-machines-be-able-to-learn-like-humans-d9160f40cdd1

5 Kaelbling, Leslie P.; Littman, Michael L.; Moore, Andrew W. (1996). "Reinforcement Learning: A Survey". *Journal of Artificial Intelligence Research*. 4: 237–285.

6 DeepMind. “AlphaGo”. Available: https://deepmind.com/research/case-studies/alphago-the-story-so-far

7 Matt Tearle. “Reinforcement Learning Onramp”. MathWorks, 2020.

8 Dmitrii Dugaev, Eduard Siemens, Ivan G. Matveev, Viatcheslav P. Shuvalov. “Adaptive Reinforcement Learning-Based Routing Protocol for Wireless Multihop Networks”. *Faculty of Electrical, Mechanical and Industrial Engineering. Anhalt University of Applied Sciences Koethen, Germany. Siberian State University of Telecommunication and Information Sciences (SibSUTIS) Novosibirsk, Russia.*

9 R. S. Sutton and A. G. Barto, “Reinforcement Learning: An Introduction” - Finite Markov Decision Processes Chapter, Second edition, in progress. 2014-2105.

10 Mnih, Volodymyr, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. “Asynchronous Methods for Deep Reinforcement Learning.” *ArXiv:1602.01783 [Cs]*, February 4, 2016. https://arxiv.org/abs/1602.01783.

11 Mathworks. “Reinforcement Learning Agents”. 2022. Available: https://www.mathworks.com/help/reinforcement-learning/ug/create-agents-for-reinforcement-learning.html

12 Lillicrap, Timothy P., Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. “Continuous Control with Deep Reinforcement Learning.” *ArXiv:1509.02971 [Cs, Stat]*, September 9, 2015. https://arxiv.org/abs/1509.02971.

13 “Bellman, R. (1957) Dynamic Programming. Princeton University Press, Princeton, NJ.- References - Scientific Research Publishing.”https://www.scirp.org/reference/ReferencesPapers.aspx?ReferenceID=2187052 (accessed Aug. 18, 2022).

14 Fujimoto, Scott, Herke van Hoof, and David Meger. "Addressing Function Approximation Error in Actor-Critic Methods". *ArXiv:1802.09477 [Cs, Stat]*, 22 October 2018. https://arxiv.org/abs/1802.09477.

15 Schulman, John, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. “Proximal Policy Optimization Algorithms.” *ArXiv:1707.06347 [Cs]*, July 19, 2017. https://arxiv.org/abs/1707.06347.

16 Mnih, Volodymyr, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. “Asynchronous Methods for Deep Reinforcement Learning.” *ArXiv:1602.01783 [Cs]*, February 4, 2016. https://arxiv.org/abs/1602.01783.

17 Schulman, John, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. “High-Dimensional Continuous Control Using Generalized Advantage Estimation.” *ArXiv:1506.02438 [Cs]*, October 20, 2018. https://arxiv.org/abs/1506.02438.

18 Schulman, John, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. "Trust Region Policy Optimization." *Proceedings of the 32nd International Conference on Machine Learning*, pp. 1889-1897. 2015.

19 Haarnoja, Tuomas, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, et al. "Soft Actor-Critic Algorithms and Application." Preprint, submitted January 29, 2019. https://arxiv.org/abs/1812.05905.

20 Mathworks. “Soft Actor-Critic Agents”. 2022. Available: https://www.mathworks.com/help/reinforcement-learning/ug/sac-agents.html

21 Jean-Claude Latombe. *Robot Motion Planning,* 2nd ed. New York, USA: Springer Science+Business Media, LLC, 1991.

22 Khatib O. “Real-Time Obstacle Avoidance for Manipulators and Mobile Robots”. *The International Journal of Robotics Research*. 1986;5(1):90-98.

23 Mathworks. “Train PPO Agent to Land Rocket - Matlab & Simulink”. 2022. Available: https://www.mathworks.com/help/reinforcement-learning/ug/train-ppo-agent-to-land-rocket.html

24 Mathworks. “Reinforcement Learning User’s Guide: Train SAC Agent for Ball Balance Control”. Available: https://www.mathworks.com/help/pdf_doc/reinforcement-learning/rl_ug.pdf

25 Mathworks. “Train TD3 Agent for PMSM Control - Matlab & Simulink”. 2022. Available: https://www.mathworks.com/help/reinforcement-learning/ug/train-td3-agent-for-pmsm-control.html

26 IBM Cloud Education. “Neural Networks”. 2020. Available: https://www.ibm.com/cloud/learn/neural-networks

27 Seminar “Introduction to Deep Learning”. UPC TelecomBCN, Barcelona (3rd Edition). 22-28 January 2020.

28 Rui Jin, Qiang Niu. “Automatic Fabric Defect Detection Based on an Improved YOLOv5”. *Mathematical Problems in Engineering.* September 2021(3):1-13. DOI: 10.1155/2021/7321394.

29 Kingma, Diederik; Ba, Jimmy (2014). "Adam: A Method for Stochastic Optimization". arXiv:1412.6980

30 Bottou, Léon (1998). “Online Algorithms and Stochastic Approximations”. *Online Learning and Neural Networks.* Cambridge University Press. ISBN 978-0-521-65263-6

31 John Pomerat, Aviv Segev, and Rituparna Datta, *On Neural Network Activation Functions and Optimizers in Relation to Polynomial Regression,* 2019 IEEE International Conference on Big Data (Big Data).

32 Zijun Zhang, *Improved Adam Optimizer for Deep Neural Networks,* ©2018 IEEE.

33 R. S. Sutton and A. G. Barto, “Reinforcement Learning: An Introduction” - Optimal Value Functions Chapter, Second edition, in progress. 2014-2105.

34 Torres J.; (2021). Introducción al aprendizaje por refuerzo profundo. Teoría y práctica en Python. Book Series. Kindle Direct Publishing. ISBN 9798599775416

Glossary

All the acronyms and their meanings were defined in their first appearance.