Contributions to the development of deep reinforcement learning-based controllers for AUV

Yoann Sola

▶ **To cite this version:**

Yoann Sola. Contributions to the development of deep reinforcement learning-based controllers for AUV. Systems and Control [cs.SY]. ENSTA Bretagne - École nationale supérieure de techniques avancées Bretagne, 2021. English. NNT : 2021ENTA0015 . tel-03901160

HAL Id: tel-03901160

https://theses.hal.science/tel-03901160v1

Submitted on 15 Dec 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE DE DOCTORAT DE

L'ÉCOLE NATIONALE SUPÉRIEURE
DE TECHNIQUES AVANCÉES BRETAGNE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Automatique, Productique et Robotique*

Par

Yoann SOLA

Contributions to the development of Deep Reinforcement Learning-based controllers for AUV

Thèse présentée et soutenue à l'ENSTA Bretagne, le 3 décembre 2021
Unité de recherche : Lab-STICC UMR CNRS 6285

Rapporteurs avant soutenance :

Vincent CREUZE	Maitre de Conference HDR, LIRMM UMR CNRS 5506, Universite de Montpellier
Jean-Philippe DIGUET	Directeur de Recherche, CROSSING, IRL CNRS 2010

Composition du Jury :

Président :	Ali MANSOUR	Professeur, Lab-STICC UMR CNRS 6285, ENSTA Bretagne
Examinateurs :	Véronique SERFATY	Responsable Innovation, Agence de l'Innovation de Défense (AID)
	Vincent CREUZE	Maître de Conference HDR, LIRMM UMR CNRS 5506, Université de Montpellier
	Jean-Philippe DIGUET	Directeur de Recherche, CROSSING, IRL CNRS 2010
	Ali MANSOUR	Professeur, Lab-STICC UMR CNRS 6285, ENSTA Bretagne
	Gilles LE CHENADEC	Enseignant-Chercheur, Lab-STICC UMR CNRS 6285, ENSTA Bretagne (Encadrant)
Dir. de thèse :	Benoît CLEMENT	Professeur, Lab-STICC UMR CNRS 6285, ENSTA Bretagne

Invité(s) :

Benoît DESROCHERS	Ingénieur, DGA Techniques Navales

Contributions to the development of Deep Reinforcement Learning-based controllers for AUV

PhD Thesis

Yoann SOLA

September 2021

Abstract

Title: Contributions to the development of Deep Reinforcement Learning-based controllers for AUV

Abstract: The marine environment is a very hostile setting for robotics. It is strongly unstructured, very uncertain and includes a lot of external disturbances which cannot be easily predicted or modelled. In this work, we will try to control an Autonomous Underwater Vehicle (AUV) in order to perform a waypoint tracking task, using a machine learning-based controller. Machine learning allowed to make impressive progress in a lot of different domain in the recent years, and the subfield of deep reinforcement learning managed to design several algorithms very suitable for the continuous control of dynamical systems. We chose to implement the Soft Actor-Critic (SAC) algorithm, an entropy-regularized deep reinforcement learning algorithm allowing to fulfill a learning task and to encourage the exploration of the environment simultaneously. We compared a SAC-based controller with a Proportional-Integral-Derivative (PID) controller on a waypoint tracking task and using specific performance metrics. All the tests were performed in simulation thanks to the use of the UUV Simulator. We decided to apply these two controllers to the RexROV 2, a six degrees of freedom cube-shaped Remotely Operated underwater Vehicle (ROV) converted in an AUV. Thanks to these tests, we managed to propose several interesting contributions such as making the SAC achieve an end-to-end control of the AUV, outperforming the PID controller in terms of energy saving, and reducing the amount of information needed by the SAC algorithm. Moreover we propose a methodology for the training of deep reinforcement learning algorithms on control tasks, as well as a discussion about the absence of guidance algorithms for our end-to-end AUV controller.

Keywords: Autonomous Underwater Vehicle, Controller, Deep Reinforcement Learning, Waypoint Tracking, Soft Actor-Critic, Proportional–Integral–Derivative

Acknowledgments

Ce manuscrit de thèse conclut une période de ma vie longue de 4 années et le moment est venu pour moi de remercier toutes les personnes qui me tiennent à cœur.

Tout d'abord je tiens à remercier l'Agence Innovation Défense (AID), la Région Bretagne et l'ENSTA Bretagne pour les financements qu'ils m'ont accordés et donc sans qui cette thèse n'aurait pas pu avoir lieu.

Ensuite je remercie du fond du cœur mon directeur de thèse Benoît Clément et mon encadrant de thèse Gilles Le Chenadec. Je les remercie d'abord pour avoir initialement modifié le sujet de thèse en 2017 en incluant au dernier moment du machine learning afin de correspondre à mes envies. Je les remercie aussi pour m'avoir guidé au travers de cette épreuve, et pour m'avoir accompagné dans une thématique qui était nouvelle autant pour eux que pour moi. Je les remercie enfin pour leur patience et leur tact concernant ma rédaction de manuscrit qui avait l'air de ne jamais en finir, je sais que je les ai grandement inquiétés par moment.

Je remercie Jean-Philippe Diguet et Vincent Creuze d'avoir accepté le rôle de rapporteur pour cette thèse de plus de 210 pages. Cela constitue une lourde tâche qui est venu s'ajouter à leurs emplois du temps déjà bien remplis. Leur rapport m'a été très utile pour améliorer toujours plus ce manuscrit.

Je remercie également Ali Mansour et Véronique Serfaty pour avoir accepté de faire partie de mon jury de thèse. Merci Ali pour tes nombreuses suggestions de correction et pour avoir présidé le jury. Pour la petite histoire c'était déjà Ali qui m'avait remis en main propre mon diplôme d'ingénieur de l'ENSTA Bretagne en 2017. Merci à Véronique et à Vincent pour avoir fait le déplacement jusqu'à Brest pour ma soutenance de thèse.

Je remercie aussi Benoît Desrochers qui n'a pas pu être présent pour ma soutenance mais qui a suivi mes travaux depuis le début en faisant parti de mon CSI.

J'adresse un remerciement particulier à Thomas Chaffre qui est arrivé en M115 bis à un

moment où j'étais perdu dans mon immense état de l'art et qui m'a permis de me recentrer sur les outils les plus importants pour notre domaine d'application. Nous avons beaucoup codé ensemble et nous avons écrit un article de conférence, et ces quelques mois de travail en commun m'ont par la suite permis de prendre mon envol et d'obtenir tous les résultats que j'ai aujourd'hui. Thomas fût donc l'élément déclencheur de tout cela, et je le remercie très chaleureusement.

Je vais maintenant énumérer nombre de doctorants, de post-doctorants et d'ingénieurs de recherche avec qui j'ai partagé d'innombrables moments formidables durant ma thèse, que ce soit dans une salle café très renommée, aux abords d'un certain lac artificiel ou dans une salle d'escalade du centre-ville de Brest :
Joris Tillet, Irène Mopin, Thomas Le Mézo, Julien Damers, Auguste Bourgois, Romain Schwab, Marie Ponchart, Fabien Novella, Maël Le Gallic, Quentin Ferdinand, Pierre Bénet, Alexandre Lefort, Thibaut Nico, Aurélie Panetier, Simon Rohou, Nathan Fourniol, Maria Luiza Costa Vianna, Alam Castillo Herrera, Carlos De Jesus Ortiz Ruiz, Benjamin Lepers, Pierre Martin, Simon Chanu, Dominique Monnet, François Cébron, Pierre Narvor, Antoine D'Acremont et Gaspard Minster.
Merci à tous et à toutes, vous furent d'excellents compagnons d'armes durant ce combat.

Je remercie également tous les autres membres du bâtiment M pour tous ces bons moments et toutes ces discussions sur les ragots concernant les élèves, sur Top Chef, sur Trackmania ou encore sur la rivalité Pays Bigouden VS Pays de Léon :
Pierre Bosser, Guillaume Sicot, Amandine Nicolle, Rodéric Moitié, Nathalie Debèse, Michel Legris, Fabrice Le Bars, Luc Jaulin, Benoît Zerr, Alain Bertholom, Didier Tanguy, Gilles Le Maillot, Hélène Thomas, Christophe Osswald et Isabelle Quidu.
Beaucoup d'entre vous ont été par le passé mes professeurs quand j'étais élève, et sont devenus par la suite des collègues géniaux et super cool. Merci pour tout.

Je remercie aussi Annick Billon-Coat, Michèle Hofmann et Patricia Cabel, pour m'avoir tant de fois aidées, voire sauvé, pour des démarches administratives, mais aussi pour tous ces agréables moments de papote quand j'allais les voir à leur bureau.

Je remercie maintenant tous mes amis hors de l'ENSTA qui m'ont permis de tenir le coup à grands renforts de ciné, de brunchs, de resto, de bars et de jeux de société:
Julie Dufrenne, Charlotte Poszwa, Dimitri Rouzo, Marion Lannuzel, Megan Quimbre, Morgane Boënnec, Pauline Labro, Armand Tanné, Nolwenn Tanné, Lambert Pêcheux et Thibauld Gianelli.
Merci les copains et les copines.

Je passe également une énorme dédicace à toute la communauté internationale de beat-

boxer. J'ai intégré cette grande famille en plein milieu de ma thèse, en allant assister à des compétitions. Je m'y suis fait de très bons amis autour d'une passion peu commune qu'est la production de musique avec sa bouche. ESH.

Je remercie et j'embrasse ma famille qui m'a vu grandir, qui m'a toujours soutenu et qui a toujours tout fait pour que je réussisse dans la vie : Patricia Sola, Jean-Gérald Sola, Mathieu Sola, Michel Sola et Marie-Françoise Sola. Nous ne nous sommes pas beaucoup vus depuis que je vis à Brest, mais je pense fort à vous tous les jours. Nous avons toutes les prochaines années pour rattraper le temps perdu, maintenant que l'on peut à nouveau voyager et que je vais avoir une bien meilleure paye pour venir vous régaler.

Je remercie et j'embrasse également ma deuxième famille qui m'a accueillie à bras ouvert et qui m'a toujours encouragée : Juliette Vyers, Rémi Vyers et Sophie Vyers. Alors que j'étais très éloigné de mes racines et de ma région d'origine, vous m'avez permis de retrouver de vrais moments de bonheur en famille et vous avez été un soutien capital durant ce périple. Merci pour tout.

Enfin je remercie et j'embrasse fort la personne la plus importante pour moi parmi tous les noms que j'ai cités précédemment : ma compagne Marie Vyers. Depuis notre rencontre en 2015, tu n'as cessé de me rendre meilleur jour après jour. Tu m'as accompagné et soutenu durant toute cette épreuve. Les 2 dernières années furent les plus difficiles de toute ma vie, et je suis passé par toute sorte d'émotions, mais tu m'as permis de ne pas craquer durant cette sombre période. Vraiment. Je pourrais traverser les plus grandes épreuves tant que je t'ai à mes côtés. Je t'aime.

Plus la lumière est forte, plus les ténèbres se doivent d'être épaisses.
Kentaro Miura

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AC: Actor-Critic

AUV: Autonomous Underwater Vehicle

BN: Batch Normalization

CNN: Convolutional Neural Network

DDPG: Deep Deterministic Policy Gradient

DL: Deep Learning

DOF: Degree Of Freedom

DQN: Deep Q-Network

DRL: Deep Reinforcement Learning

EKF: Extended Kalman Filter

GNC: Guidance-Navigation-Control

LfD: Learning from Demonstration

LNN: Lyapunov Neural Network

LOS: Line-Of-Sight

LQR: Linear–Quadratic Regulator

MDP: Markov Decision Process

MIMO: Multi-Input-Multi-Output

ML: Machine Learning

MLP: Multilayer Perceptron

MPC: Model Predictive Control

NN: Neural Network

PER: Prioritized Experience Replay

PID: Proportional–Integral–Derivative

PIDNN: Proportional–Integral–Derivative Neural Network

PG: Policy Gradient

PPO: Proximal Policy Optimization

RL: Reinforcement Learning

RNN: Recurrent Neural Network

ROV: Remotely Operated underwater Vehicle

SLAM: Simultaneous Localization and Mapping

SAC: Soft Actor-Critic

SD: Standard Deviation

SISO: Single-Input-Single-Output

TD: Temporal-Difference

TD3: Twin Delayed Deep Deterministic

TRPO: Trust Region Policy Optimisation

UAV: Unmanned Aerial Vehicle

USV: Unmanned Surface Vehicle

UUV: Unmanned Underwater Vehicle

Chapter 1

Introduction

Controlling a robotic platform in a marine environment is a particularly challenging task, since it is a very hostile environment. It is strongly unstructured, meaning that the lack of the structure makes the environment difficult to model. Moreover this environment includes a lot of uncertainties and external disturbances that cannot be easily predicted or modelled: the wind, the waves on the surface, the ocean currents, the seabed topography, the potential presences of objects, fishes, or rocks, etc. Another specific problem found in marine robotics is the lack of positioning in the sea or the ocean, since the GPS signals cannot be propagated in the water. Without a valid or accurate model of the environment, the control task is more difficult and the controllers become harder to tune. Finally, as in a lot of other control tasks, all the input, output and state signals include random noises, due to the environment or the robot itself.

1.1 Contextual elements

Robotics is a vast domain, and mobile robotics is one its subfield [158], where the studied robots are able to move by themselves, either on the ground, in the air or in the water. Therefore marine robotics [235] is itself a subfield of mobile robotics [305]. Various robotic platforms can be the object of marine robotics [374]. Here is an exhaustive list of the marine robots which can be found in the literature:

- **Autonomous Underwater Vehicles (AUV)** are small submarines able to act by themselves [282]. They are usually operated by thrusters and fins. Since this work focus exclusively on the control of AUV, they will be discussed in details in section 4.1. They can also be called Unmanned Underwater Vehicle (UUV) in the literature.
- **Remotely Operated underwater Vehicles (ROV)** are mini-submarines piloted remotely by a human, at the contrary of AUVs [372][74]. A cable is used in order to send

the input signals and to supply energy.

- **Unmanned Surface Vehicles (USV):** are simply autonomous boat, controlled by an algorithm and not a human [367][311].
- **Autonomous sailboats:** are sailboat operated by an algorithm [315][160]. Servomotors are usually used to control the rudder and the sail in order to benefit from the wind. Sometimes an emergency propeller can be found.
- **Underwater gliders:** are marine devices propelled by variable-buoyancy propulsion instead of real thrusters. The depth is controlled by ballasts, as in real submarine, and then this vertical movement allow the glider to move forward thanks to its design based on side wings [112][161].

We will focus on AUVs and we will not discuss the other marine robots. Their development began in the 50's and has not stop to evolve throughout the decades, by continuously improving the mechanical design, the actuators, the sensors, the electronics, the communication, the power supply and the control algorithms. Their uses have also diversified: from research (hydrography, oceanography) and military applications (communication/navigation network nodes, mine countermeasures), to commercial and hobby uses.

In this work, we chose to compare a machine learning-based controller and a classical controller on a waypoint tracking task performed by an AUV. This control task is generalizable to a large amount of marine robotics missions, since every path can be decomposed into successive waypoints to follow. Every missions where trajectories need to be followed or specific targets needs to be reached can be reduced to a waypoint tracking task composed of one or more waypoints.

Machine learning

Machine learning is a subfield of the area of artificial intelligence. It is located at the intersection of applied mathematics, neurobiology and computer science. This family of algorithms was born in the late 50's, with the term "machine learning" created by Arthur Samuel in 1959 [285]. The perceptron algorithm is one of the building block of machine learning, and later deep learning, and was created in 1958 by Frank Rosenblatt [276]. Another milestone was the creation of the backpropagation algorithm created in the 80's [280], which led to a second wave of interest in machine learning by researchers. The most impressive achievements happened very recently with a third wave of interest starting in the 2010's, initiated in 2012 by the advances made by the Google Brain team [198] and the AlexNet architecture [184]. This third wave of interest and its recent results have been made possible thanks to the great improvements in computational power (especially with the use of Graphics

Processing Units) coupled with the great availability of training data needed by the algorithms (mainly thanks to the web and the democratization of the Internet of Things).
Throughout the decades, machine learning algorithms have been successfully applied to various application domains: computer vision [296][345], speech recognition [69][68], robotics [240][262], video games [41][300], biology [328][356], data mining [228][361], etc. The field of machine learning is composed of three main subdomains: supervised learning, unsupervised learning and reinforcement learning.

Supervised learning tasks [52][123] are based on a dataset containing vectors of data called inputs $\mathbf{x}$, and a scalar associated with each vector called the label $\mathbf{y}$. The input $\mathbf{x}$ is composed of descriptive features x_i. The task is divided in two phases:

- The training phase: the goal of the algorithm is to learn to match the inputs $\mathbf{x}$ to their corresponding label $\mathbf{y}$.
- The test phase: the algorithm must retrieve the label $\mathbf{y}$ corresponding to new inputs $\mathbf{x}$ that were not used during the training phase.

If the label is discrete, it corresponds to a number of classes and the supervised learning task is a classification; if the label is continuous, it corresponds to the value of a given parameter and the supervised learning task is a regression. Different metrics and error measurements are used depending on the nature of the task.
An example of supervised learning task is the recognition of handwritten digits [201][178], where the inputs $\mathbf{x}$ are vectors containing the pixel values of images representing handwritten digits and the label is the digit that corresponds to this image, from 0 to 9. This is a classification task of supervised learning where the algorithm has to identify digits that were written by humans.

In **unsupervised learning**, the dataset is not labeled and the algorithm has to gather similar data in order to build classes [23][54]. The degree of similarity of data is defined using specific metrics for each task. This is often called as clustering [7].
An example of unsupervised learning use case is recommender systems [58][377], which learn to recommend products that are similar to the websites visited by a user.

While supervised learning and unsupervised learning tasks make use of very similar paradigms, **Reinforcement Learning (RL)** tasks [320] use a completely different paradigm and should be considered separately from the two previous machine learning approaches. This approach uses concepts from neuroscience.
RL tasks are based on the principle of "trials and error" and on the Markov Decision Process (MDP) paradigm: an agent evolves in an environment by taking actions and receives an observation of the state of the environment and a reward signal. The received reward is

positive if the action taken by the agent was a good choice at the moment, and is negative if it was a bad choice. The reward signal and the state sent to the agent depends of each task. The goal of the agent is to maximize the long-term sum of all rewards it receives over time. At the beginning, the agent takes random actions in order to explore the environment, and then exploits the knowledge gathered during its interactions with the environment.
An example of RL task is an agent controlling a video game character in order to obtain the best score [334]. The state of the environment is the set of the pixels displayed by the video game and the reward signal would be positive for each point obtained and negative for each life lost by the character.

Moreover, other kinds of machine learning approaches can be found in the literature, depending the nature of the problems, such as a semi-supervised learning [381][382] which deals with partially labeled sets of data for example.
The type of predictions outputted by the machine learning algorithm can also be grouped into two categories:

- **Classification:** The output is discrete and called a class. The algorithm has to predict to which class each input belongs to. When the task involves only two classes of data, it is called a binary classification task. When more than two classes are found, the term multiclass classification is preferred. It is important to note that a multiclass classification task can be transformed into several independent binary classification tasks [36].
- **Regression:** The output is continuous and the algorithm has to estimate the value of one of more variables.

Figure 1.1 shows two simple examples of supervised learning tasks. The left plot shows a binary classification task represented in the space of the descriptive features x_i that compose the input vector $\mathbf{x}$. Here the inputs are only composed of two descriptive features x_1 and x_2 which allow to show the entire task on a 2D plot. The black line represents the decision boundary of the learning algorithm, which shows the seperation that the algorithm is making between the two classes. From the algorithm's point of view, all data found on the same side of the decision boundary are labeled in order to belong to the same class.
The right plot shows a simple regression made where the input is composed of only one scalar x. The horizontal axis represents the input x and the vertical axis represents the label y. The black line shows the model that is computed by the learning algorithm in order to fit as closely as possible the dataset. This model will allow the algorithm to predict labels y from any input data x, coming from the dataset or not.
The distinction between a classification and a regression is independent from the subfield needed by the the task: for example, a classification can be supervised or unsupervised.

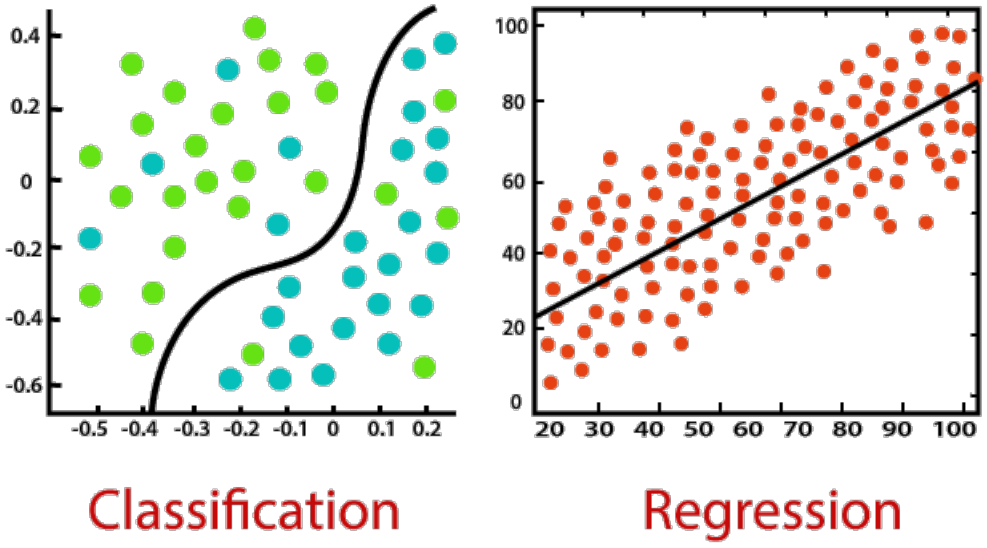


Figure 1.1. The difference between classification and regression tasks.

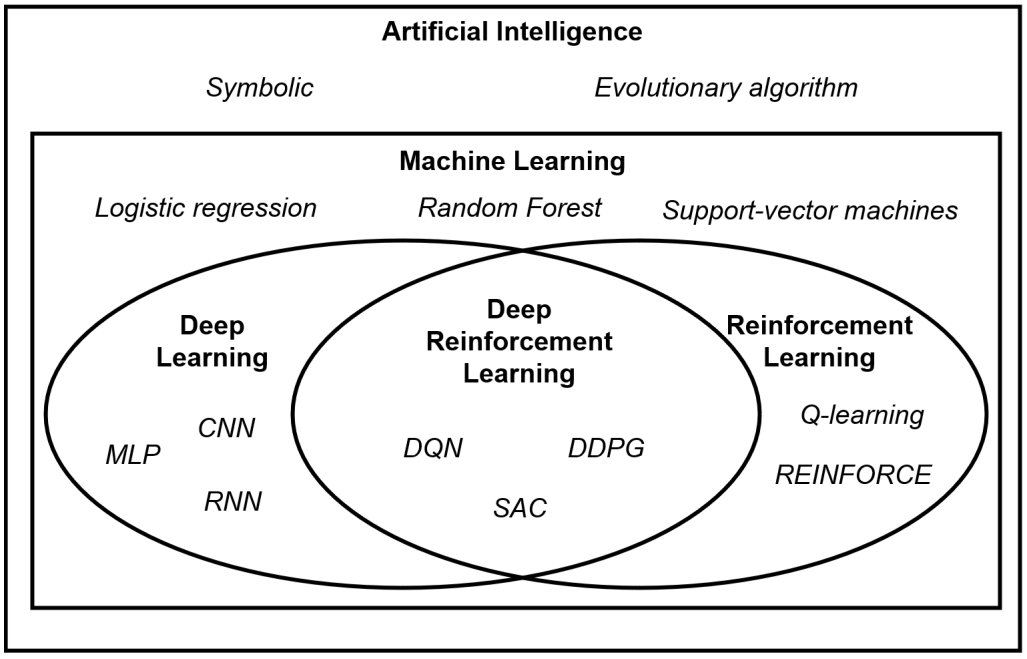


Figure 1.2. The links between Artificial Intelligence, Machine Learning and Deep Learning.

Over the past few years, the growth of the field of machine learning has led to the multiplication of technical terms: the distinction needs to be made between machine learning, deep learning and reinforcement learning. The Figure 1.2 shows how these subfields are connected to one another. It also cites names of famous algorithms (some of them will be explained in this work).
First of all, machine learning is contained within the field of artificial intelligence, and deep learning and reinforcement learning are contained within machine learning. Machine learning can also be categorized as part of the data science field and the term *data scientist* can be used in order to name the person deploying, experimenting or even improving machine learning algorithms. In scientific papers, the term *machine learning* often refers to all classical techniques used in data science that are not based on deep neural networks: logistic regressions, decision trees, support vector machines, k-nearest neighbors, k-means, etc. The term *traditional machine learning* can be used in order to distinguish these algorithms from deep learning and reinforcement learning.
Deep learning refers to the supervised and unsupervised tasks that are solved using deep neural networks. This subfield will be detailed in section 3.1.
RL encompassed all algorithms used to solve tasks modeled by MDPs. These techniques will be described in section 3.2. However, the increasing complexity of reinforcement learning tasks led reasearchers to implement deep learning tricks and tools inside reinforcement learning algorithms, leading to the subfield of Deep Reinforcement Learning (Deep RL). Depending on the case, specific terms can be added in order to emphasize the preponderant aspects of the machine learning problem: safe learning, imitation learning, incremental learning, hierarchical learning, etc.

Finally, we have to also make a distinction between the parameters and the hyperparameters of a machine learning model. The *parameters* refer to the variables that will be updated during the training phase in order to fit the data or to fulfill the task. Finding the right parameters is the objective of all machine learning tasks, since they will defined the behaviour of the trained model. They are often updated using optimization algorithms applied to a given objective function.
The *hyperparameters* refer to all constants representing the design choices made for a model: they allow to define the structure of the algorithm, to control the learning process and to define the goals of the task. While the parameters are updated by the learning process during the training phase, the hyperparameters are tuned manually during the implementation of the machine learning algorithm. They can be chosen by either trials and error, or using common rules found in the literature. It is also worth mentioning the recent works dealing with the automatic selection of the hyperparameters, which led to the creation of the subfield of Automated Machine Learning (AutoML) [146][325].

In this work, we choose to use a state-of-the-art deep reinforcement learning algorithm

called the Soft Actor-Critic (SAC) [116] (described in section 3.2.3.3), in order to control an Autonomous Underwater Vehicle (AUV) and to compare it with a PID controller, a classical control theory algorithm.

Control theory

Control theory is an applied mathematical subfield studying the dynamical systems. The term *automation* can also be found in the literature. A dynamical system is a mathematical object allowing to represent a concrete or abstract phenomenon. The goal of control theory is to study the behaviour of dynamical systems and to eventually synthesize a controller. A controller is an algorithm defined by several parameters and allowing to fulfill a control task. The dynamical system is defined by a mathematical model called the *plant* and evolves in a given environment. The plant model can be linear or nonlinear, and can include model uncertainties. The plant and the controller form together a *feedback system*. More details concerning all these definitions will be explained in chapter 2.
Various family of techniques appeared in the field of control theory throughout the history. The development needs of control theory followed the trends dictated by the different historical periods: the industrial revolutions, the wars, the Space Race, the economic globalization, etc. The main control theory approaches are:

- **Classical control theory** is the first set of methods created in control theory and is the basis of the next approaches [87]. The PID controller [162] is the most known algorithm from this period (detailed in the section 2.2).
- In **adaptive control**, the parameters of the controller are tuned automatically, in order to adapt the algorithm to the current situation of the plant or the environment [327][15].
- **Stochastic control** techniques have to deal with uncertainties found in the plant model and with random noises included in different signals of the whole feedback systems. They adapt probabilistic methods to the field of control theory [190][11].
- Like for stochastic control, **robust control** approaches have to deal with the model uncertainties [120] and noises, but also with the rejection of external disturbances. They make use of frequency domain analysis and stability, analysis [73][380]. Two well-known robust control algorithms are the sliding mode control [77][261] and the H-infinity control [337][366].
- In **optimal control,** the control problem is viewed as an optimization problem with a criterion (called the *cost*) which must be minimized or maximized. Mathematical constraints can also be added to the optimization problem [342][209]. The most common optimal control techniques are the Model Predictive Control (MPC) [91][236] and Linear-Quadratic-Gaussian (LQG) control [16].

- The **fuzzy logic** is an approach used in a lot of domains other than control theory [249]. It uses the concept of partial truth, and is opposed to the Boolean Logic in which the variables can only be true or false. In fuzzy control systems, the input signals are analyzed as having continuous values between 0 and 1 [194][19].
- **Hierarchical Control Systems (HCS)** are based on multiple devices and software components which are sorted as a hierarchical tree [6][83].
- Some of the previous methods cannot be applied to nonlinear systems. **Nonlinear control** [307][151] refers to all the techniques allowing to deal with them, and can sometimes include elements from the previous techniques. For example, a nonlinear controller can be made with the use of feedback linearization [313][204].

Even if control theory is an older field than machine learning, some of these approaches are still able to design state-of-the-art controllers. All machine-learning-based controllers should be compared to these baselines on specific control tasks in order to be qualified as suitable to control and robotics tasks.

1.2 Objectives of this work

The purpose of this work is the control of Autonomous Underwater Vehicles (AUV) in order to perform waypoint tracking tasks. Two types of controllers will be compared on these tasks: PID controllers and deep reinforcement learning-based controller. The chosen deep reinforcement learning algorithm will be the Soft Actor-Critic (SAC), one of the most recent machine learning algorithm. Several goals will need to be fulfilled, from either a control theory or a machine learning perspective. We made the choice to compare the SAC algorithm with the PID controller, because PID is the most used controller in both the academic field and the industry.

The main objective of this work will be to evaluate if the SAC is able to understand the dynamics of the AUV and to learn how to fulfill the control task, based only on the received data. The SAC-based controller will have to reach target waypoints, while dealing with disturbances such as varying ocean currents or sensors and actuators noises. This controller will try to achieve the end-to-end control of the AUV: it will simultaneously fulfill high-level and low-level control functions, meaning that no guidance algorithm will be implemented. This will be opposed to the PID control approach, which is associated with a simple straight lines guidance method.

If the SAC algorithm is able to converge towards a satisfactory behaviour during its learning process, one of the secondary objectives will be to evaluate if the SAC can outperform

the PID controller on various control criteria such as the percentage of waypoints reached by the AUV during the tests, the amount of deviation of the AUV from its trajectory (due to the ocean currents) or even the power consumption.

From a machine learning perspective, one goal of this work will be to study how the configuration of the various parameters of the SAC algorithm can impact the performance of the controller. We will also try to analyse if the use of advanced training techniques can improve the results. This will allow to propose a methodology dealing with the training of deep reinforcement learning algorithms on waypoint tracking tasks. We will try to make this procedure as generalizable as possible to other marine robotics tasks.

Lastly, all of our tests will be performed in simulators, but we want the controllers to be easily transferable to a real-world AUV. We will have to carefully choose the best practical tools, but also to tune them in order to be as realistic as possible.

1.3 Thesis organization

In this section, we detail the structure of this PhD thesis, as well as the logic followed by the successive chapters and sections.

Chapters from 2 to 5 correspond to the state of the art and the review of the existing literature. We tried to give an extensive overview of each domains by tackling as many topics as possible. The reader should be aware that the notations used in all these chapters will be unified as far as possible in order to keep consistency, so a few of the variables will be renamed differently from their original papers. Moreover some notions from control theory and machine learning can have similar names, so we will used different notations to named them in order to better differentiate these two fields.
Chapter 2 presents the principles of the control theory. It defines the basic notions of dynamical systems, feedback systems and the Guidance-Navigation-Control approach, before explaining how PID controllers work. It also introduces the notions of stability of dynamical systems and Lyapunov functions.
Chapter 3 describes the principles of machine learning. It especially focuses on all the algorithms and the elements which led to the Soft Actor-Critic (SAC). It starts with deep learning by describing the artificial neural networks, the Adam optimizer and the batch normalization algorithm. The remaining of the chapter concern reinforcement learning and deep reinforcement learning. The basic paradigm of Markov Decision Process is explained in details, followed by the Temporal-Difference learning approach. The policy gradient section lists all the building blocks of the SAC and ends with a complete presentation of this algorithm.
Chapter 4 combines the presentation of the Autonomous Underwater Vehicles (AUV) and

a selection of the existing works of machine learning applied to control theory and AUVs. The mathematical models used to represent AUVs will be detailed, as well as the existing approaches in guidance, navigation and control for AUV. The machine learning section has been divised into subsections: one for deep learning and the other for reinforcement learning.
Chapter 5 deals with safe reinforcement learning, a subfield of reinforcement learning seeking to provide safety guarantees to the learning process. These techniques are classified into two groups: the methods based on the modification of the classical cost function used in reinforcement learning and those based on the modification of the exploration process of the algorithms. Some of these approaches mix elements of control theory with reinforcement learning.

Chapter 6 corresponds to our main contributions to fulfill the objectives defined in the introduction. It starts with the successive presentations of the simulation tools, the AUV (the RexROV 2, a cube-shaped six degrees of freedom robot) and the control task we used in our work, followed by the description of the two evaluated controllers and the definition of the metrics allowing to assess their performance. Then we show the results we get during our initial trials on the waypoint tracking task (these early results were published in [309]).
The next sections detail our first main contribution, which is an in-depth study of how the changes in the configuration of the state vector can affect the performance of the SAC algorithm. This study is carried out on a simplified task in order to try to outperform the PID controller. We managed to match the PID controller in terms of the number of reached waypoints, while outperforming it on the energy consumption. Moreover we managed to get these results with reduced state vectors, which means than AUV can fulfill its missions with less sensor information.
Our second main contribution consists in the experimentation of advanced training techniques: the Batch Normalization algorithm taken from deep learning and the Learning from Demonstration approach taken from safe reinforcement learning. Batch Normalization managed to improve the performance of the SAC algorithm and to stabilize the learning process. We tried to make the SAC learn the task from demonstrations provided by PID controllers, but this approach did not allow the deep reinforcement learning algorithm to learn the task and to understand the dynamics of the AUV. We proposed a training methodology as well as a discussion concerning the use of deep RL techniques for the end-to-end control of AUVs.

The chapter 7 is the conclusion of the thesis. The first part summarizes all the results and allows to make the link between our proposals and the existing state of the art. It answers the questions raised in the introduction about the objectives we defined. The second part describes the main ideas about what we expect to do in the future in order to continue this work.

Appendices A to C describe further elements that have been removed from the state of the art in order to not overload the reading. Appendix A makes a quick presentation of traditional machine learning and describes the logistic regression, an algorithm belongings to the origins

of the neural networks. Appendix B shows additional techniques used in deep learning such as regularization, initialization and several optimizers. Appendix C presents different deep reinforcement learning algorithms belonging to the Temporal-Difference learning and policy gradient approaches.

Chapter 2

Elements of the control theory

In this chapter, we will describe the main principles of the control theory. We will explain various concepts related to dynamical systems and feedback systems: modeling, control, guidance, navigation and stability. We will also present the well-known PID controller. A lot of these concepts will be used in sections 4 and 5.

2.1 Feedback systems

Feedback systems is one of the most fundamental building block of almost all control schemes. The textbook [14] is a very complete reference to start with and to go through the main concepts.

2.1.1 Dynamical systems

A *system* is an abstract object used for modelling an ensemble of elements interacting with each other and following specific rules. It is a general notion used in a variety of scientific and non-scientific domains [344]. It is often represented by multiple equations or functions involving variables from several interacting elements. The system is evolving in an *external environment*. A *dynamical system* is a system modeled by functions taking into account the time t [167] and whose variables evolve over time.

In order to better illustrate the different components of dynamical systems, we will use the example of an inverted pendulum [40][157]: it is composed of a cart able to move towards the left or the right, with a pendulum fixed to its top by a rigid rod. The rod is able to rotate thanks to a pivoting link and the control task is to keep the pendulum upright by moving the cart horizontally.

2.1.1.1 The components of a dynamical system

In the control theory literature, the following notation is used in order to name the different elements of the dynamical system:

- The vector $\mathbf{x}$ is called the *state* of the system. It includes the internal variables describing the current state of the system. For example the state vector of the inverted pendulum is composed of the horizontal position and the linear speed of the cart and by the angle and the rotational speed of the rod.
- The vector $\mathbf{u}$ is called the *input* of the system and represents the variables allowing the environment to interact with the system. In the case of the inverted pendulum, the input vector is only composed of the horizontal force applied to the cart, generating the move of the cart.
- The vector $\mathbf{y}$ is called the *output* or the *measure* of the system and is composed of the variables of the system observed by the external environment. The output vector of the inverted pendulum is formed by the horizontal position of the cart and the angle of the rod.

Depending on the dynamical system, the state, input and output vectors can share variables, as it is the case with the state and output vectors of the inverted pendulum: both of them include the position of the cart and the angle of the rod. Moreover, the dimension of the input and output vectors defined the dynamical system as a *single-input-single-output (SISO) system* or a *multi-input-multi-output (MIMO) system.*

2.1.1.2 The different representations of a dynamical system

A dynamical system can be represented either in the temporal domain or in the frequency domain [14]. Both of these representations are equivalent and we can switch from one representation to the other using mathematical operations.

The temporal domain representation is composed of an ensemble of differential equations involving the different elements of the dynamical system:

$$\mathbf{F}(t,\ \mathbf{x}(t),\ \mathbf{u}(t),\ \mathbf{y}(t))\ =\ 0 \tag{2.1}$$

where $\mathbf{F}(.)$ is a set of several differential equations implying the state $\mathbf{x}$, the input $\mathbf{u}$ and the output $\mathbf{y}$. These equations are called the *dynamic equations.*
Depending if the differential function $\mathbf{F}(.)$ are linear or non-linear, the dynamical system is said to be *linear* or *non-linear.* Moreover, if the differential equations $\mathbf{F}(.)$ do not involve directly the time t (the vectors $\mathbf{x}$, $\mathbf{u}$ and $\mathbf{y}$ are time-dependent, but if the variable t is not

found in the equations $\mathbf{F}(.)$), the dynamical system is said to be *time invariant*.

Another temporal representation can be derived from the dynamic equations 2.1 (given for a general dynamical system):

$$\begin{cases} \frac{d\ \mathbf{x}(t)}{dt} &= \mathbf{f}(t,\ \mathbf{x}(t),\ \mathbf{u}(t)) \\ \mathbf{y}(t) &= \mathbf{h}(t,\ \mathbf{x}(t),\ \mathbf{u}(t)) \end{cases} \tag{2.2}$$

where $\mathbf{f}(.)$ and $\mathbf{h}(.)$ are general non-differential equations. These equations are called *state equations* and composed the *state space representation* of the system. More specifically the first one is called the *evolution equation* and the second one is the *observation equation*.
In the case of a Linear Time Invariant (LTI) dynamical system, the state equations 2.2 become:

$$\begin{cases} \frac{d\ \mathbf{x}(t)}{dt} &= \mathbf{A}\ \mathbf{x}(t) &+ &\mathbf{B}\ \mathbf{u}(t) \\ \mathbf{y}(t) &= \mathbf{C}\ \mathbf{x}(t) &+ &\mathbf{D}\ \mathbf{u}(t) \end{cases} \tag{2.3}$$

where $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ and $\mathbf{D}$ are real-valued matrices.

In the frequency domain, a compact description of the input-output relation called the *transfer function* can be found from the state equations 2.3 of a LTI system, by using *Laplace transforms* [94]:

$$\frac{\mathbf{Y}(s)}{\mathbf{X}(s)} \ = \ \mathbf{G}(s) \ = \ \mathbf{C}(s\mathbf{I} - \mathbf{A})^{-1}\mathbf{B} \ + \mathbf{D} \tag{2.4}$$

where $\mathbf{X}(s)$ and $\mathbf{Y}(s)$ are the Laplace transforms of $\mathbf{x}(t)$ and $\mathbf{y}(t)$ respectively, $\mathbf{G}(s)$ is the transfer function, s is a complex number and $\mathbf{I}$ is the identity matrix.

Both the state space representation and the transfer function allow to carry out advanced control design and specific analysis of dynamical systems that would not have been possible inside the temporal domain [14][104].

2.1.2 Closed-loop dynamical systems

The goal of control theory is to design a *controller* able to make the output signal of a dynamical system follow a given signal called the *reference*. In order to follow this reference, the controller measures the current value of the output signal at any time step. This is done by making a loop from the output of the controller to the input of the controller, forming a *closed-loop dynamical system* or a *feedback system*.

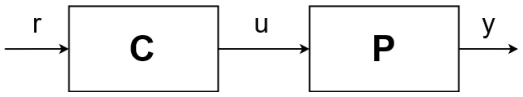


Figure 2.1. An open loop control system.

2.1.2.1 Block diagrams and feedback signals

In control theory, block diagrams are often used in order to describe a control architecture. An example of block diagram is shown in the Figure 2.1: **P** is the transfer function of the dynamical system that is being controlled, often called the *plant* in the control theory literature; **C** is the transfer function of the controller and is itself considered as a dynamical system with its own input and output signals; r, u and y are respectively the reference signal, the input of the plant and the output of the plant. These signals can be multidimensional in the case of MIMO systems (as defined in 2.1.1).
The controller **C** is trying to make the output signal y of the plant **P** follow the reference r. The output signal corresponds to the controlled variables of the plant and constitute a measure of the system, while the reference is the desired setpoint that the output variables of the plant need to reach. These reference setpoints can be specified by a human or another algorithm. The controller achieves the given control task by computing the input needed by the plant: the output of **C** is the input of **P**.
As explained in section 2.1.1, the transfer functions are a view of dynamical systems belonging to the frequency domain. This means that the signals r, u and y are frequency representations of their respective variables and must respect a certain structure [14]: they belong to a class of time functions of the form $X(s)e^{st}$, with $X(s)$ being a complex function and s a complex number. These signals depend on the complex variable s (which will not be written for the signals r, u and y for more clarity). The signals y and u can be written based on Figure 2.1 as:

$$\begin{cases} y &= \mathbf{P}.u \\ u &= \mathbf{C}.r \end{cases} \tag{2.5}$$

The plant of Figure 2.1 does not send a feedback signal or measure to the controller. This control system is called an *open-loop system* or a *feedforward system*. The controller has no information about the plant and does not know if the output is correctly following the reference. The controller is computing the input according to specific plans made in advance. The control system is insensitive to measurement noises and cannot introduce a risk of instability if the plant is already stable (an instability is a divergence in the computations of the input). However, an open-loop system is very sensitive to model uncertainties [14][44].

Figure 2.2 shows a *closed-loop system* or a *feedback system*. Instead of only taking the reference r into account, the input of the controller **C** becomes the difference between the reference r and a measure of the output y. The controller is now able to adapt the computation

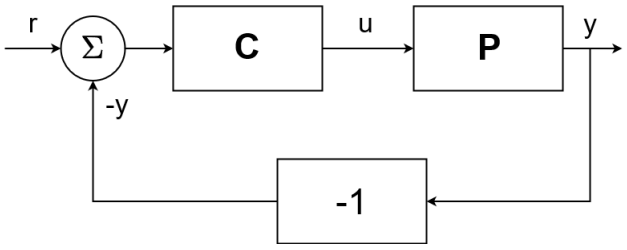


Figure 2.2. A close loop control system.

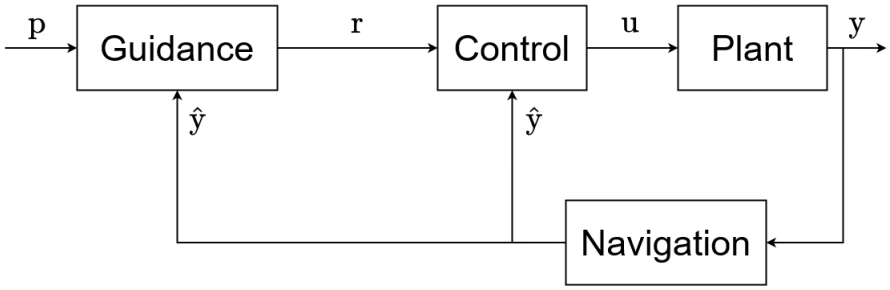


Figure 2.3. A general Guidance-Navigation-Control (GNC) control.

u according to a deviation of y from r, and is not performing planning anymore.
The output y can now be written as:

$$\begin{cases} y &= \mathbf{P}.u \\ u &= \mathbf{C}.e &= \mathbf{C}.(r-y) \end{cases} \tag{2.6}$$

with $e = r$ - y being called the *tracking error*.

Closed-loop systems are less disturbed by model uncertainties than open-loop systems (since the transfer function $\mathbf{P}$ is no longer found in the computation of u in equation 2.6), but are more sensitive to measurement noises, since the equation of u is now based on the measurement of y. Moreover, a risk of instability exists, since large tracking error signals could cause too large inputs u for the plant $\mathbf{P}$.

2.1.2.2 Guidance-Navigation-Control systems

In control theory, a *Guidance-Navigation-Control (GNC)* system is the name of the global hierarchy of algorithms structuring the control of a dynamical system [78][263]. A general GNC system is represented on Figure 2.3. Each block represents an independent system or algorithm. The components of a GNC system are the following:

- **The plant system:** as previously, the plant is the system which needs to be controlled.

Its output signals y are the variables being controlled. The plant system used in this work will be detailed in section 4.1.

- **The navigation algorithm:** the role of the navigation components is to estimate the output y of the plant system named $\hat{y}$. Sometimes the output cannot be directly determined and an algorithm is therefore required to perform the estimation: when the measures are very noisy (filtering), when some information are missing (interpolation), etc. This component is also called a *state observer* and is said to perform a *state estimation* [22][353]. The Extended Kalman filter [136][163], regression techniques [121][189], the Bayesian estimation [110][259] or the interval analysis [159][174] are common examples of navigation approaches.

- **The control algorithm:** often called the controller, its function is computing the input u of the plant [245]. This input is based on the difference between the estimate $\hat{y}$ of the output (given by the navigation component) and a reference r (given by the guidance component). This component can also be referred as the *low-level controller*.

- **The guidance algorithm:** the guidance component is in charge of generating the reference r. It takes into account the current estimate of the output, $\hat{y}$, and several parameters p specified in advance (the parameters can be either variable or fixed). The guidance algorithm allows to fulfill different goals [158]: path planning [196], obstacle avoidance [172], waypoint tracking [70], etc. These different goals can be combined during a same control task, and a trade-off must be found between each of them. Line-of-Sight techniques [20][322], artificial potential fields methods [195][243], Voronoi diagrams [33][72] are examples of well-known guidance approaches. The guidance system can also be called the *high-level controller*.

Each component of a GNC system can be considered as a separate dynamical system, with its own inputs, outputs, state variables. All the components can also be reduced to one dynamic system encapsulating all of them, depending on the view needed for the control task. The whole GNC system forms a closed-loop dynamical system.

2.2 PID controllers

A GNC system includes a controller component. His role is to make the output y of the plant closer to the reference r computed by the guidance algorithm, based on the estimated output $\hat{y}$ given by the navigation algorithm. Different approaches may be considered in order to compute the appropriate input u needed by the plant, depending on the needs of the control task. A selection of control approaches has already been presented in the introduction.

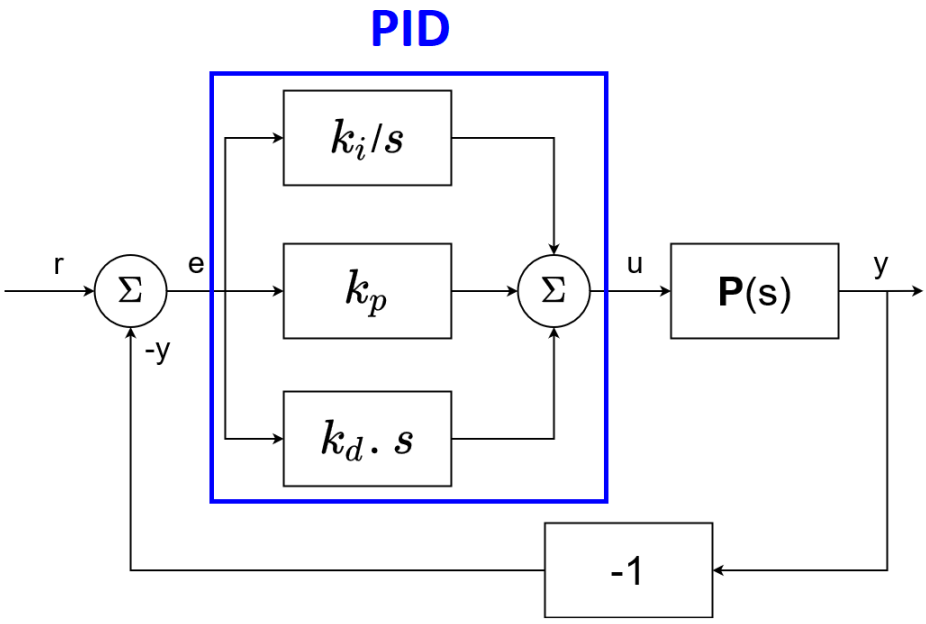
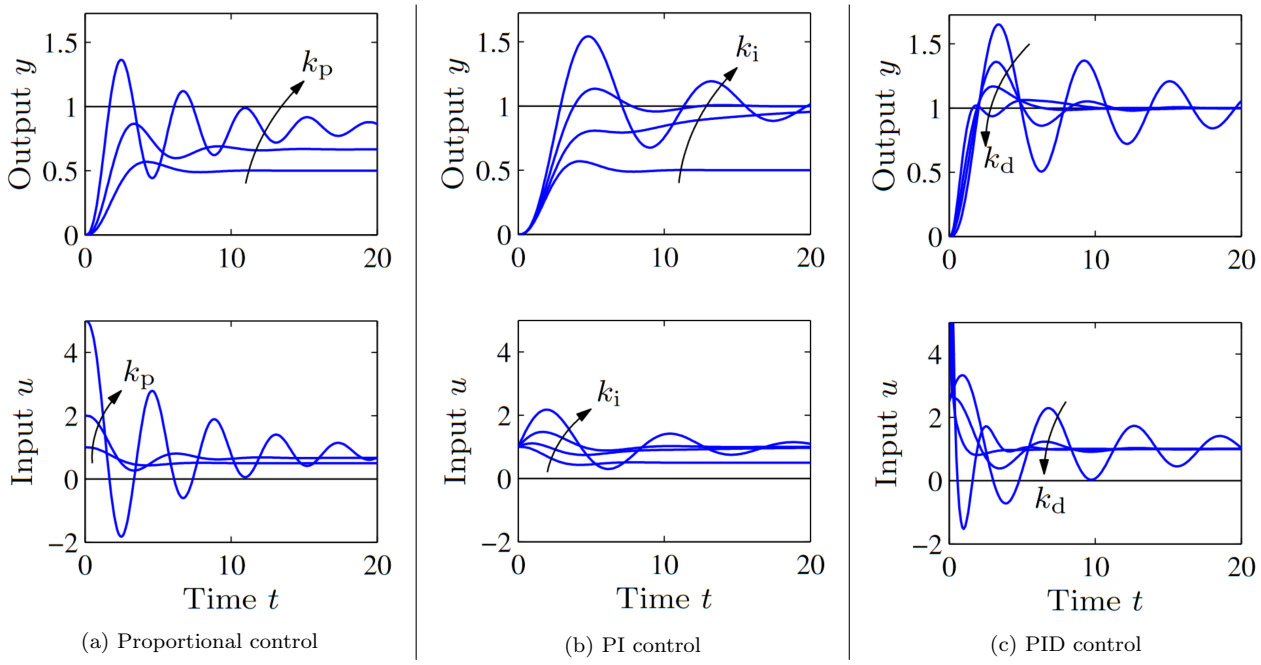


Figure 2.4. The block diagram of a feedback system composed of a PID controller and a plant model.

The Proportional–Integral–Derivative (PID) controller is the most used controller in both the academic field and the industry [12][14] and belongs to the classical control theory approach. The input u needed by the plant system is computed by the PID thanks to the tracking error $e(t) = r(t) - \hat{y}(t)$, with $r(t)$ being the reference and $\hat{y}(t)$ being the estimated output of the plant. The equation used by the PID controller is the following:

$$u(t) \;=\; k_p\, e(t) \;+\; k_i \int_0^t e(\tau)d\tau \;+\; k_d \,\frac{de(t)}{dt} \tag{2.7}$$

where k_p, k_i and k_d are real-value scalars (or matrices in the case of multi-dimensional signals of MIMO systems) called the *gains*.
As shown in equation 2.7, the PID controller takes his name from the fact that it computes the input u proportionally to three distinct terms: the tracking error (representing the past), the integral of the tracking error (representing the present) and the derivative of the tracking error (representing the future). The block diagram of Figure 2.4 shows a typical PID controller in the frequency domain (with the use of Laplace transforms), without guidance and navigation components.



(a) Proportional control (b) PI control (c) PID control

Figure 2.5. Responses to step changes in the reference value r for a system with a proportional controller (a), PI controller (b), and PID controller (c). The reference varies at the time step 0 from the value 0 to the value 1. The process has the transfer function $P(s) = 1/(s+1)^3$, the proportional controller has parameters $k_p = 1$, 2, and 5, the PI controller has parameters $k_p = 1$, $k_i = 0$, 0.2, 0.5, and 1, and the PID controller has parameters $k_p = 2.5$, $k_i = 1.5$, and $k_d = 0$, 1, 2, and 4.

The gains k_p, k_i and k_d of the PID controller need to be specifically tuned for each plant system and each control task. Figure 2.5 shows the influence of the value of each gains (example taken from [14]). It shows the signal input u computed by the PID controller and the output y and allows to note the following behaviours:

- the proportional gain k_p affect the speed of convergence of the plant system: the bigger k_p is, the faster the system converge to the reference.
- the integral gain k_i corrects the *static error*, which is the difference between the input and the output of a system when the time converges to the infinity: here it is a measure of the deviation of the output from the reference, taken asymptotically. The bigger k_i is, the closer the output is to the reference.
- the derivative gain k_d acts on the magnitude of the oscillations: the bigger k_d is, the smaller the oscillations of the output are.

These gains can be either constant (fixed by control theory methods [182][338]), or variable. In the latter case, they will be updated automatically during the execution of the control task

according to a criterion or an external algorithm [13][47]. They can also be tuned using the frequency domain [203].

The PID controller is easy to be implemented, as only equation 2.7 is needed. It is also easy to tune since it has just three parameters, which can eventually be tuned empirically by interpreting the resulting signals (like in the example of Figure 2.5). However in some cases, even a self-tuning PID controller is not enough, and advanced approaches may be considered. Several limitations can be found in the PID control approach [17][318]. A PID controller can have difficulties with the systems involving specific non-linearities or varying internal parameters. It can also lack of responsiveness in the presence of large low-frequency disturbances. Finally the tuning of the PID controller must be carefully chosen, since a trade-off must be found between the regulation abilities and the response time of the controller.

2.3 Stability of feedback systems

One of the first analysis operated on a dynamical system is a stability analysis [191]. This helps to understand the system being controlled and provide guidelines on the choice and the tuning of the controller. Some of the following elements will be used by several safe reinforcement learning approaches of the section 5. We will describe only several well-known techniques.

2.3.1 Equilibrium points

As defined in section 2.1.1, $\mathbf{x}(t)$ is the state of a dynamical system. Let us note:

$$\frac{d\ \mathbf{x}(t)}{dt} = \mathbf{F}(\mathbf{x}(t)) \tag{2.8}$$

with $\mathbf{F}$ being a general function. It is very similar to the evolution equation found in equation 2.2. When the state $\mathbf{x}(t)$ of a dynamical system varies over time, we say that the system follows a given *trajectory* in the *statespace*.
One of the first things to carry out when analyzing the stability of a dynamical system is to find its *equilibrium points* x_e, which can be computed by resolving the equation:

$$\mathbf{F}(x_e) \ = \ 0 \tag{2.9}$$

These equilibrium points are specific points of the state space of a dynamical system, where the dynamics remain stationary [14]. It means that the state variables will not diverge to extreme values. The values of the equilibrium points are often used by controllers as reference signals in order to stabilize the dynamical system, letting it in a steady behaviour.

In practice, the equilibrium points can also be used in order to linearize complex non-linear systems [183][312]. Mathematical approximations can be performed around a given equilibrium point, and the resulting linear system is a valid approximation of the original non-linear system in the vicinity of the equilibrium point. The new linear system allows to perform a simpler stability analysis.

2.3.2 Pole placement method

One of the most known stability analysis approach is the pole placement method [192][362]. This method can only be applied to linear systems.

Let $\mathbf{G}(s)$ be the transfer function of a linear system. The pole placement method rewrites this transfer function using a partial fraction decomposition:

$$\mathbf{G}(s) = k \, \frac{(s - z_1) \dots (s - z_n)}{(s - p_1) \dots (s - p_m)} \tag{2.10}$$

where k is a real number, $(z_1 \dots z_n)$ the complex roots of the numerator of $\mathbf{G}(s)$ and $(p_1 \dots p_m)$ the complex roots of the denumerator of $\mathbf{G}(s)$. $(z_1 \dots z_n)$ are called the *zeros* of $\mathbf{G}(s)$ and $(p_1 \dots p_m)$ are called the *poles* of $\mathbf{G}(s)$.

A linear system is said to be stable if all the poles of its transfer function have strictly negative real parts. The system will remain close to its nearest equilibrium point.
This definition can be used to certified the stability of a controller, by taking the transfer function of the whole linear feedback system (composed of the controller and the plant). The name *pole placement* refers to the fact that the tuning of the parameters of the controller must place the poles of the transfer function of the whole feedback system in the negative part of the complex plane.

2.3.3 Stability with PID controllers

The stability of a feedback system composed of a PID controller can be guaranteed by multiple methods [253][293], but we will only cite two different approaches.

One convenient method is to find stability regions in the parameter space of the PID controller [237]. It corresponds to regions in the spaces of value of the gains k_p, k_i and k_d (which can be multi-dimensional in the case of MIMO systems) where the whole feedback system meets specific frequency domain criteria. The PID gains are then bounded by these stability regions: they are the sets of all PID gains that stabilize the closed-loop system. Figure 2.6 (taken from [237]) shows an example of stability regions the gains of a PD controller

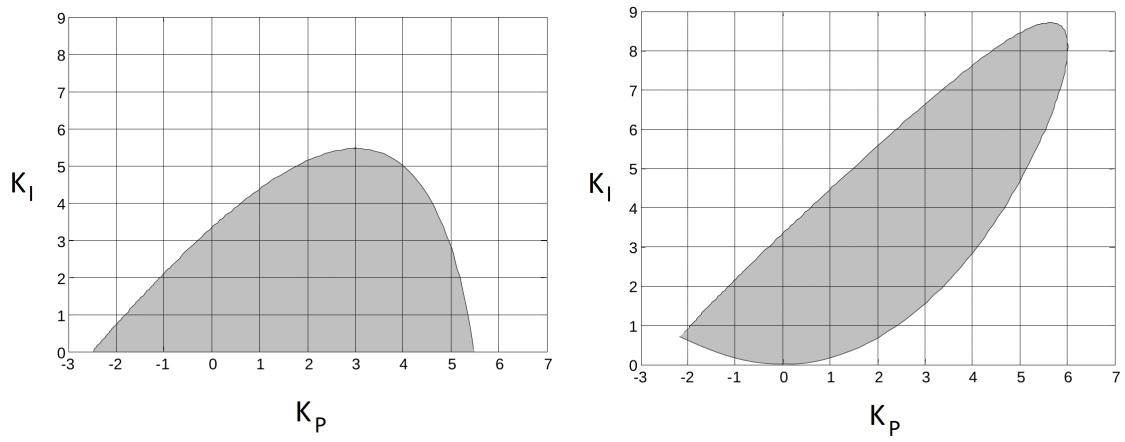


Figure 2.6. Examples of regions of stability for a PI controller (on the left) and a PID controller (on the right).

and a PID controller.

The pole placement method from the previous section can also be applied to PID controllers [351][383].

2.3.4 Lyapunov functions

The Lyapunov stability analysis is one of the most powerful stability guarantee tool, since it can be applied to any nonlinear dynamical system [171][220].

A *Lyapunov funtion* is an energy-like function V of $\mathbb{R}^n \rightarrow \mathbb{R}$ which can be used to determine the stability of a general nonlinear dynamic system:

$$\frac{dx}{dt} = \mathbf{F}(x) \quad \text{with } x \in \mathbb{R}^n \tag{2.11}$$

The stability of this system around an equilibrium point is guaranteed by using the following theorem [14]. This theorem is formulated for an equilibrium taken at the origin of the state space ($x_e = 0$) for simplicity, but it can be generalized to any equilibrium point x_e.

Lyapunov stability theorem: Let V be a function on $\mathbb{R}^n$ and let $\dot{V}$ represent the time derivative of V along trajectories of the system dynamics (2.11):

$$\dot{V} = \frac{\partial V}{\partial x}\frac{dx}{dt} = \frac{\partial V}{\partial x}\mathbf{F}(x) \tag{2.12}$$

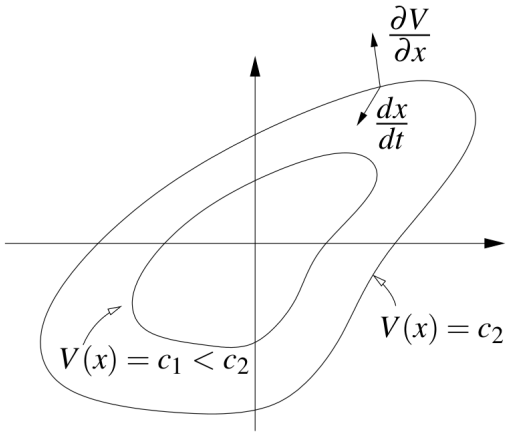


Figure 2.7. A two dimensions Lyapunov function

Let $B_r = B_r(0)$ be a ball of radius r around the origin of the state space of the system.
If there exists $r > 0$ such that V is positive definite and $\dot{V}$ is negative semidefinite on B_r, then $x_e = 0$ is (locally) stable in the sens of Lyapunov: x remains in the vicinity of x_e.
If V is positive definite and $\dot{V}$ is negative definite in B_r, then $x_e = 0$ is (locally) asymptotically stable: x converge to x_e as the time tends to infinity.
B_r defines what is called a *region of attraction*, since the trajectories of the system will stay inside this region of the state space [65][341].

The Lyapunov function is a real-valued function: the smaller its value is, the closer the defined region of attraction will be to the equilibrium point. An example of Lyapunov functions computed in a two dimensional state space is shown on the Figure 2.7 (taken from [14]).

A *phase portrait* is a representation of the evolution of the trajectories of a dynamical system. A vector field is plotted in the state space of the system and a dot or a curve is attributed to each initial conditions of the system.
An example of a phase portrait of an inverted pendulum is shown on Figure 2.8 (taken from [14]). The blue lines and the blue arrows show the evolution of the trajectories over time. We can see multiple equilibrium points (where the trajectories converge towards a same point) and a Lyapunov function is drawn in red.

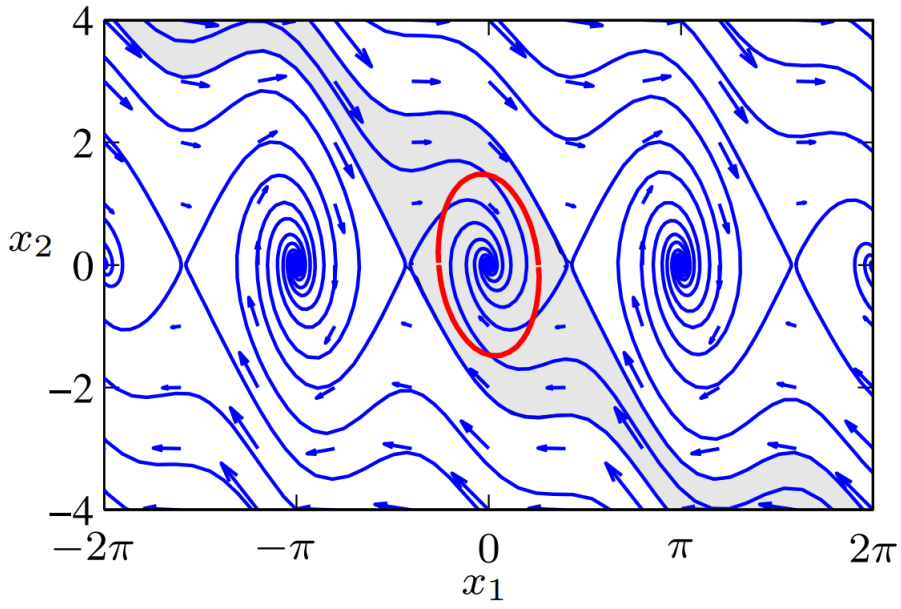


Figure 2.8. A phase portrait of an inverted pendulum with the red line delimiting the region of attraction covered by a Lyapunov function.

Lyapunov functions are one of most the powerful tool for guaranteeing the local stability of a dynamical system, but they are very hard to use, since no systematic method can be applied. Indeed, the Lyapunov functions need to be found empirically for each dynamical system (no general rule exist for deriving them), which is often not possible for complex systems such as robots. Significant progress has been made in order to found computational methods for these Lyapunov functions, [103][38]. However, these methods cannot often be applied to complex dynamical systems.

In the remaining of this work, the control theory elements presented in this chapter will be either reused by machine learning approaches or compared with machine learning algorithms.

Chapter 3

Elements of Deep Reinforcement Learning

We will explain in this chapter key concepts of deep learning and reinforcement learning. We have also added appendix A in order to present additional elements of traditional machine learning. Moreover additional explanations about deep learning and reinforcement learning can be found respectively in the appendices B and C. In this chapter, we only kept the essential parts to understand the algorithm Soft Actor-Critic we will use during the results of the chapter 6.

3.1 Artificial neural networks and Deep learning

As mentioned in the introduction, the term *Deep Learning* [109][199] refers to the supervised and unsupervised tasks that are solved using deep neural networks. The major difference between the traditional machine learning and the deep learning approaches is that deep learning requires a very large amount of data in order to estimate or classify well: a task requiring several millions of data samples is very common. Traditional machine learning algorithms such that random forest or k-nearest neighbors can work even with small datasets. Another difference is that deep learning algorithms take raw data in input, whereas traditional machine learning approaches require a feature extraction step. The feature extraction step is carried out using specific knowledge from the application area of the task, in order to filter the data. This step is implicitly included in the deep learning techniques, which thus does not require an expert from the application area.

In this section, we will only focus on the deep learning algorithms used for supervised tasks. The book [109] is one of the best entry points for deep learning researchers.

3.1.1 Multilayer Perceptron

The works on *Artificial Neural Networks (ANN)* algorithms, often shortened to *Neural Networks (NN)*, have begun in the 40's [176][224]. This family of techniques takes its name from the domain of neuroscience and aims to imitate the way the human brain works.

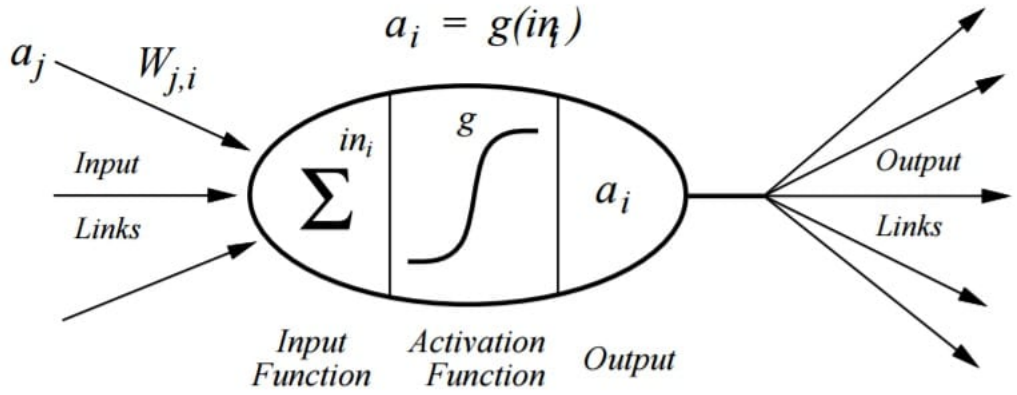
Multilayer perceptrons (MLP) [123] are the most common type of ANNs for supervised learning tasks. A MLP is constructed from multiple *perceptrons* (also called an *artificial neuron*). A representation of a perceptron is shown on Figure 3.1 (taken from [21]). The implementation of a perceptron is very close to the logistic regression algorithm, described in the appendix section A.1.5. It is composed of two parameters which need to be learnt: the weight vector $\mathbf{w}$ and the bias b. The biais b may be renamed as W_0 and may be included in the weight vector $\mathbf{w}$; like on Figure 3.1. Figure 3.1 shows the i-th perceptron of a MLP, this is why the subscript i is used with all the components of the perceptron.
After multiplying the weight vector $\mathbf{w}$ with the entry vector of the perceptron, a non-linear function called the *activation function* is applied to the result of this multiplication (the activation function is called g on Figure 3.1). Several well-known activation functions are used in the deep learning community:

- The sigmoid function: $y(x) = \frac{1}{1+\mathrm{e}^{-x}}$
- The hyperbolic tangent function: $y(x) = \frac{\mathrm{e}^{x}-\mathrm{e}^{-x}}{\mathrm{e}^{x}+\mathrm{e}^{-x}}$
- The Rectified Linear Unit (RELU) function: $y(x) = \max(0, x)$
- The leaky ReLU function: $y(x) = \max(0.01x, x)$

A multilayer perceptron is composed of several perceptrons connected all together, as represented on Figure 3.2 [1]. Hereinafter, we call a perceptron as a neuron, which is the name more commonly found in the deep learning community. The MLP is organized in layers:

- An *input layer* simply takes the data matrix $\mathbf{X}$ in input and sends it directly to next layer. The matrix $\mathbf{X}$ has the size (number of descriptive features, number of training data).
- Several *hidden layers*, each one is composed of multiple neurons. The MLP shown on the Figure 3.2 is composed of only one hidden layer of four neurons, but neural networks can be composed of more hidden layers. The neurons of each layer can have different activation function: the hidden layer of the MLP from the figure 3.2 is composed of hyperbolic tangent neurons, but it could for example be followed by a hidden layer of

[1] Taken from *Neural Networks and Deep Learning*, https://www.coursera.org/learn/neural-networks-deep-learning?specialization=deep-learning



$$a_i = g(\sum_j W_{j,i} a_j)$$

Figure 3.1. A perceptron, the base unit of deep learning.

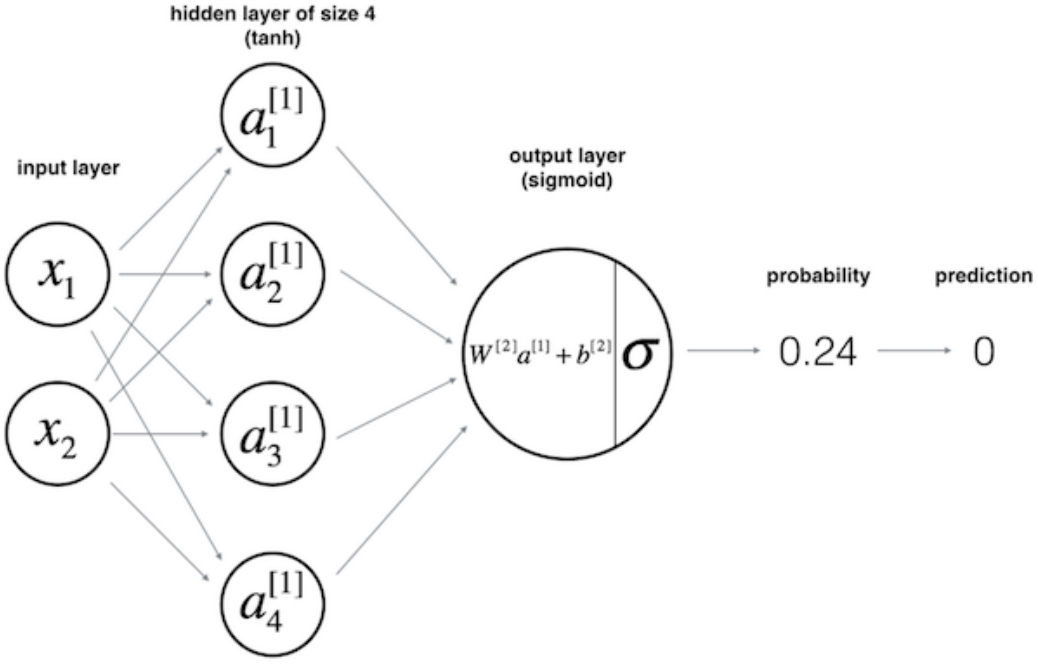


Figure 3.2. A multilayer perceptron (MLP)

ReLU neurons.
The principle of hidden layers is that the output of each neuron is connected to the inputs of all neurons of the following layer. Moreover, the neurons are not connected to the neurons found in the same layer. MLP can also be called *fully connected neural network*, since each neuron is connected to all the neurons of previous and next layers.
The role of the hidden layers of the network is to learn and represent the links between all the descriptive features of the matrix $\mathbf{X}$. This representation is used in order to take decisions and to solve a given problem.

- The output layer computes the predicted output $\hat{y}$ of the MLP. If the task is a classification, $\hat{y}$ corresponds to the class of input $\mathbf{X}$. If the task is a regression, $\hat{y}$ is the estimation of a variable computed from the input $\mathbf{X}$. $\hat{y}$ can be either a scalar or a vector.
  For binary classification tasks, the output layer is composed of only one neuron implementing the sigmoid function. This output ranges from 0 to 1: if the output is strictly inferior to 0.5, the predicted label corresponds to the first class; if the output is superior to 0.5, the predicted label corresponds to the second class.
  For multiclass classification tasks, the number of the neurons composing the ouput layer will be equal to the number of classes. Each neuron represents a class and implements the *Softmax function* [36]: it allows to output the probability of the class represented by the neuron to be the true class of the input $\mathbf{X}$. The final estimated output of the neural network is then chosen by selecting the neuron with the maximum output, thus taking the most likely class according to the probabilities outputted by each neuron.
  For regression tasks, the output layer is composed of one or more neurons with no activation function: the expected ouputs do not need to range from 0 to 1. If the output values are supposed to be positive, the ReLU function can be used.

The layers have an assigned number, from 0 for the input layer to N for the output layer. Each layer i has two parameters which needs to be learnt or updated:

- A weight matrix $\mathbf{W_i}$ of size (number of neurons in layer i, number of neurons in layer i-1), which corresponds to the aggregated weight vectors of all the neurons from that layer.
- A bias vector $\mathbf{b_i}$ of size (number of neurons in layer i, 1), which corresponds to the aggregated bias parameters of all the neurons from that layer.

These parameters allow to compute the outputs of all the neurons of a same layer at the same time. For a given layer i, we have:

$$\mathbf{a_i} \;=\; \text{activation}\left(\mathbf{W_i}^T\,\mathbf{a_{i-1}} \;+\; \mathbf{b_i}\right) \tag{3.1}$$

with *activation* being the activation function of the layer i (applied element-wise) and $\mathbf{a_i}$ being the vector corresponding to the output of the layer i and of size (number of neurons in

layer i, number of training data).

The input layer is the only layer without parameters $\mathbf{W_i}$ and $\mathbf{b_i}$, since its purpose is just to take data matrix $\mathbf{X}$ in input and to send it to the first hidden layer, without any modifications. Several hyperparameters need to be choosen in order to set the architecture of the neural network: the number of hidden layers, the number of neurons per layer, the activation function of each layer.
When the MLP has only one or two hidden layers, it is called a *shallow neural network*. When it has more than two layers, the term *deep neural network* is used.
The activation functions used in the neural network needs to be non-linear. If linear activation functions are implemented, the whole neural network can be reduced to a simple linear combination $\mathbf{W'}^T\mathbf{X} + \mathbf{b'}$ and its modeling ability will not be powerful enough: the neural network will become a simple linear regression.

3.1.2 Training neural networks

We are going to describe how to train a neural network like the MLP in order to learn a ML task.

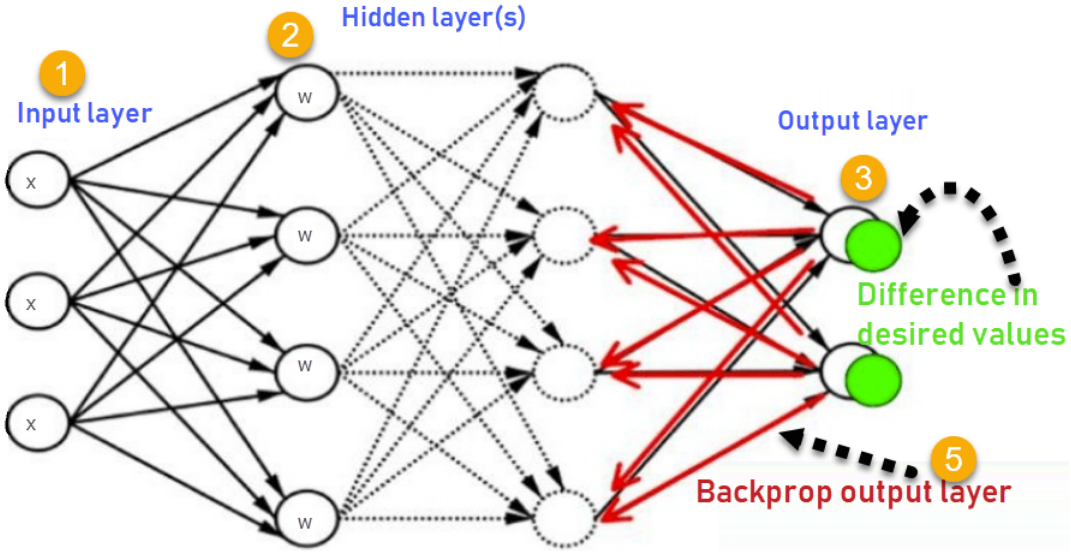


Figure 3.3. The backpropagation process in neural networks.

3.1.2.1 Backpropagation

The backpropagation is an algorithm used to train the neural networks [200][280]. The steps operated by the backpropagation are represented on Figure 3.3 [2]:

1. The input layer takes data matrices $\mathbf{X}$ in input.

2. The input layer is sent to the first hidden layer and equation 3.1 is applied. The same equation is applied successively to all the following layers, until the output layer giving the estimated label or estimated variables in the case of a regression task. This is called the *forward propagation*.

3. A *loss function* $L(\hat{y}, y)$ is used in order to measure the error between the predicted outputs $\hat{y}$ of the neural network and the true labels y of the training set. The loss function needs to be chosen depending on the nature of task. For binary and multiclass classification tasks, the *cross-entropy* loss function [241] function will be used. For a regression task, the *Mean Absolute Error* (MAE, also called *L1 loss*) [359] or the *Mean Squared Error* (MSE or also called *L2 loss*) [206][346] will be used. Other loss functions can be found in the litterature: the hinge loss [62][71], the Huber loss [142], the Kullback–Leibler divergence [187][188], etc.
   A cost function J is then used in order to know the global classification or regression error on the whole dataset. It simply computes the mean of the loss function values obtained from all the outputs of the forward propagation:

$$J(\mathbf{W}, \mathbf{b}) = \frac{1}{m} \sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right) \tag{3.2}$$

   with m being the number of data samples contained by the training set. The cost function gives a measure of the performance of the neural network and helps to know when the training process needs to be ended.

4. The parameters $\mathbf{W_i}$ and $\mathbf{b_i}$ of each layer i (except the input layer) need to be updated in order to minimize the cost function:

$$\min_{\mathbf{W_i}, \mathbf{b_i}} J(\mathbf{W_i}, \mathbf{b_i}) \quad \forall\, i \in [1, N] \tag{3.3}$$

   with N being the number of layers of the neural network, apart from the input layer.

[2] Taken from *Back Propagation Neural Network: What is Backpropagation Algorithm in Machine Learning?*, https://www.guru99.com/backpropogation-neural-network.html

The update of the parameters is made thanks to the *gradient descent*, a very well-known optimization algorithm [64][207]. It is applied to the parameters as follows:

$$\begin{cases} \mathbf{W_i} = \mathbf{W_i} - \alpha \frac{\partial J(\mathbf{W_i}, \mathbf{b_i})}{\partial \mathbf{W_i}} \\ \mathbf{b_i} = \mathbf{b_i} - \alpha \frac{\partial J(\mathbf{W_i}, \mathbf{b_i})}{\partial \mathbf{b_i}} \end{cases} \quad \forall i \in [1, N] \tag{3.4}$$

with α being the *learning rate*, a hyperparameter allowing to control the speed and the precision of the convergence towards the minimum of the cost function.

The partial derivatives are matrices of the same size as their corresponding parameter:

- $\frac{\partial J(\mathbf{W_i}, \mathbf{b_i})}{\partial \mathbf{W_i}}$ has the size (number of neurons in layer i, number of neurons in layer i-1).
- $\frac{\partial J(\mathbf{W_i}, \mathbf{b_i})}{\partial \mathbf{b_i}}$ has the size (number of neurons in layer i, 1).

This part of the learning process corresponds to the *backpropagation* step.
The partial derivatives of equation 3.4 are computed using the chain rule:

$$\frac{\partial J(\mathbf{W_i}, \mathbf{b_i})}{\partial \mathbf{W_i}} = \frac{\partial J(\mathbf{W_i}, \mathbf{b_i})}{\partial \mathbf{a_i}} \cdot \frac{\partial \mathbf{a_i}}{\partial \mathbf{W_i}} \tag{3.5}$$

with $\mathbf{a_i}$ being the output vector of all the neurons of the layer i. Each partial derivative vector has the same size and the multiplication found in equation 3.5 is applied element-wise.

The partial derivative $\frac{\partial J(\mathbf{W_i}, \mathbf{b_i})}{\partial \mathbf{a_i}}$ is itself computed using the chain rule:

$$\frac{\partial J(\mathbf{W_i}, \mathbf{b_i})}{\partial \mathbf{a_i}} = \frac{\partial J(\mathbf{W_i}, \mathbf{b_i})}{\partial \mathbf{a_{i+1}}} \cdot \frac{\partial \mathbf{a_{i+1}}}{\partial \mathbf{a_i}} \tag{3.6}$$

Each partial derivative is computed using terms from either the same layer or the next layer, which are incrementally computed during the backpropagation. Linear algebra is used in order to vectorize these chain rules and matrix operations allow to found each term of the previous chain rules. More details of these computations can be found in [109].

As the arrow suggests on Figure 3.3, the chain rules are first applied to the output layer, then the values of the computed partial derivatives are propagated backwards through all hidden layers. The same chain rules are applied to the bias vectors $\mathbf{b_i}$.
This backpropagation allows each hidden layer to update its parameters $\mathbf{W_i}$ and $\mathbf{b_i}$ one

after another, until reaching the first hidden layer.

In practice, the gradient descent used during the backpropagation gives decent results, but more advanced optimization algorithm can be used.

3.1.2.2 The Adam Optimizer

The gradient descent mentioned in the previous section is not the only used optimization algorithm in deep learning. More advanced optimization algorithms, called *optimizers*, can be employed depending on the needs of the task. The choice of the right optimizer can be viewed as an additional hyperparameter. A given optimizer can be the best choice during one task, regarding the performances it provides, while performing poorly on another task.

The optimizer called *Adam* was defined in [175] and means *Adaptive moment estimation*. The idea of the authors of [175] was to combine the principles of the gradient descent with momentum and RMSprop, two optimizers described in section B.3. The idea of these two optimizers is to apply a moving average to either the partial derivatives (for the gradient descent with momentum), or the square of the partial derivatives (for RMSprop). The results of these moving averages will be used in order to update the parameters, instead of the raw partial derivatives.
In the case of the Adam optimizer, the gradient descent update equations 3.4 become:

$$\begin{cases} \text{MA}_{\mathbf{W_i}} = \beta_1 \text{ MA}_{\mathbf{W_i}} + (1-\beta_1) \frac{\partial J(\mathbf{W_i},\mathbf{b_i})}{\partial \mathbf{W_i}} \\ \text{MA}_{\mathbf{b_i}} = \beta_1 \text{ MA}_{\mathbf{b_i}} + (1-\beta_1) \frac{\partial J(\mathbf{W_i},\mathbf{b_i})}{\partial \mathbf{b_i}} \\ \text{SMA}_{\mathbf{W_i}} = \beta_2 \text{ SMA}_{\mathbf{W_i}} + (1-\beta_2) \left(\frac{\partial J(\mathbf{W_i},\mathbf{b_i})}{\partial \mathbf{W_i}}\right)^2 \\ \text{SMA}_{\mathbf{b_i}} = \beta_2 \text{ SMA}_{\mathbf{b_i}} + (1-\beta_2) \left(\frac{\partial J(\mathbf{W_i},\mathbf{b_i})}{\partial \mathbf{b_i}}\right)^2 \\ \mathbf{W_i} = \mathbf{W_i} - \alpha \frac{\text{MA}_{\mathbf{W_i}}}{\sqrt{\text{SMA}_{\mathbf{W_i}} + \epsilon}} \\ \mathbf{b_i} = \mathbf{b_i} - \alpha \frac{\text{MA}_{\mathbf{b_i}}}{\sqrt{\text{SMA}_{\mathbf{b_i}} + \epsilon}} \end{cases} \quad \forall i \in [1, N] \tag{3.7}$$

MA stands for the *Moving Average* and implements an exponentially weighted moving average of the partial derivative of the cost function J with respect to both parameter vectors $\mathbf{W_i}$ and $\mathbf{b_i}$ of the *i-th* layer of the neural network. SMA stands for *Squared Moving Average* and is implementing an exponentially weighted moving average of the square of the partial

derivative of the cost function J with respect to the parameter vectors $\mathbf{W_i}$ and $\mathbf{b_i}$ of the *i-th* layer of the neural network. β_1 and β_2 are two hyperparameters allowing to weight the update of the moving averages. α is the same learning rate as in the gradient descent. N is the number of layers of the neural network. ϵ is a small real number avoiding the denominators of the fractions to be equal to zero. The authors of [175] suggested the following values: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$. The learning rate α still need to be specifically tuned for each deep learning task.

Adam is the most used optimizer in the deep learning community and is often the most effective one, at the exception of some very specific tasks. It allows to speed up the learning process thanks to the smoothing of the partial derivatives performed by the moving averages. The Adam optimizer is often applied to reduced sets of training data instead of the whole available dataset. This reduced set are called *mini-batch* and are successively sampled from the training set. When the whole training set has been sampled trough successive mini-batches, we said that an *epoch* has been performed. The mini-batches are an additional way allowing to speed up the learning process.

3.1.2.3 Batch normalization

The inputs of the dataset are usually normalized during the preprocessing of the data, by substracting the mean to all the input vectors and then dividing the results of these subtractions by the variance. The mean and the variance are computed on all the data of the dataset. This allows to shape the cost function (in the parameter space) in order to reach the minimum point faster.

The *Batch Normalization (BN)* algorithm [150][340] applies the same idea to all the layers of the neural network. The inputs of all the layers are normalized before applying the activation function. For a given layer i, the normalization is applied to $\boldsymbol{z}_i = \mathbf{W_i}^T \mathbf{a_{i-1}} + \mathbf{b_i}$. The BN algorithm can be applied in the context of mini-batches and epochs, or not. This form of normalization allows to speed up the learning process. Moreover, it can also stabilize the training of the neural networks: during the backpropagation, the BN algorithm allows to prevent the partial derivatives to explode to extremely high or extremely low values. The pseudo-code of the BN algorithm is detailed below and is described for a given layer l:

---

1. For each mini-batch or batch of size m, we have $\boldsymbol{Z}_l = [\boldsymbol{z}_l^{(1)}, \ldots, \boldsymbol{z}_l^{(m)}]$, where $\boldsymbol{z}_l^{(i)}$ is the the pre-activation vector of the l-th layer given for i-th data sample.
2. Computation of the vector:

$$\mathbf{mean} = \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{z}_l^{(i)} \tag{3.8}$$

3. Computation of the vector:

$$\mathbf{variance} = \frac{1}{m}\sum_{i=1}^{m} (\boldsymbol{z}_l^{(i)} - \mathbf{mean})^2 \tag{3.9}$$

4. Computation of the normalized pre-activation vector:

$$\boldsymbol{Z}_l^{\text{norm}} = \frac{\boldsymbol{Z}_l - \mathbf{mean}}{\sqrt{\mathbf{variance} + \epsilon}} \tag{3.10}$$

where ϵ is a small real number avoiding the division by 0. The division by $\sqrt{\mathbf{variance} + \epsilon}$ is aplied element-wise to the numerator.

5. (Optionnal) If necessary, the pre-activation can be forced to belong to another distribution, with a different mean and a different variance:

$$\boldsymbol{Z}_l^{\sim} = \boldsymbol{\gamma}^T . \boldsymbol{Z}_l^{\text{norm}} + \boldsymbol{\beta} \tag{3.11}$$

where $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are vectors and can be additional learnable parameters.

The BN algorithm is applied in this way to all the layers of the neural network.

3.2 Reinforcement learning

As defined in the introduction, Reinforcement Learning (RL) is one of the three main subfields of Machine Learning, alongside with Supervised Learning and Unsupervised Learning. We are going to describe the main elements allowing to understand the Soft Actor-Critic, the RL algorithm we will use in the chapter 6. One of the main entry points for researchers is the well-known book [320]. A taxonomy of the main Reinforcement Learning algorithms is presented on the Figure 3.4 (taken from [375]), showing the main families of algorithms of RL. Some of these approaches will be tackled in the following sections.

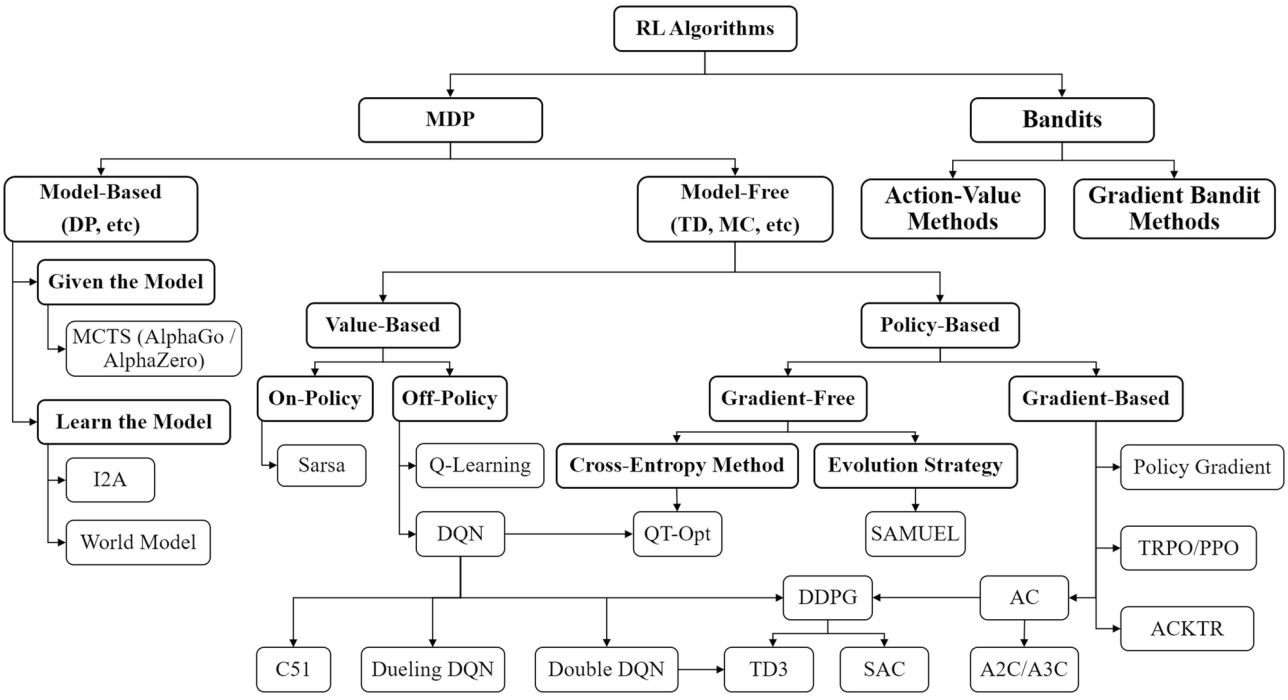


Figure 3.4. A taxanomy of the most known Reinforcement Learning algorithms

3.2.1 The paradigm of Markov Decision Process

The main formalization of Reinforcement Learning (RL) tasks is based on the paradigm of Markov Decision Process (MDP) [25]. A MDP is a discrete-time stochastic control process used in optimization problems falling into the subfield of dynamic programming [138].

3.2.1.1 The agent

As represented on Figure 3.5 (taken from [320]), a MDP is composed of an agent trying to achieve a goal defined by the task. The agent evolves in an environment and learns the task by trial and error.

At time step t, the agent is in a given state S_t of the environment and performs an action A_t. The state and the action can be vectors. The environment responds to the agent by sending to it an observation of the new state S_{t+1} and a scalar signal called the *reward* R_{t+1}. The role of the reward is to judge the actions carried out by the agent with respect to the states of the environment: if the reward is positive, the action was good to carry out in the given state in order to complete the task; otherwise, it was a bad action to perform and it moved the agent away from its goal. The whole sequence of all the visited states and all the actions taken by the agent during T steps is called a *trajectory*:

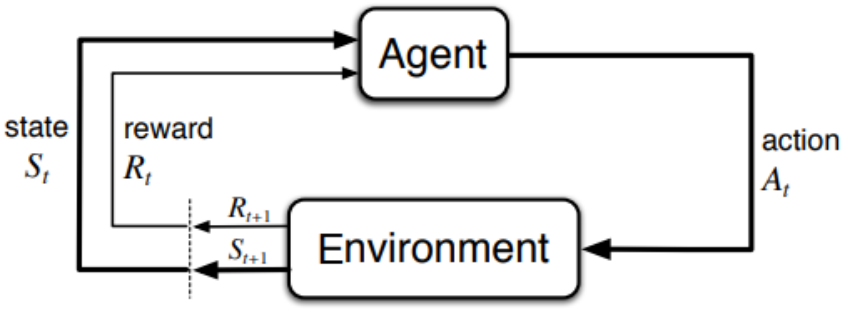


Figure 3.5. The paradigm of Markov Decision Process

$$S_1, A_1, R_2, S_2, A_2, \ldots, S_T \tag{3.12}$$

The goal of the agent is to maximize the long term sum of all the received rewards. In section 3.1, the cost function was the key element defining the supervised learning task. In reinforcement learning, the task is mainly defined by the reward function and must be tuned carefully.

The behaviour of the agent is defined by a policy $\pi(a|s)$, allowing it to choose which action to perform in a given state. The policy can be stochastic or deterministic: deterministic policies generate a single action $\pi(s) = a$, while stochastic policies generate a vector composed of the probabilities of choosing each possible actions, also called the probability distribution over actions $\pi(a|s) = \mathbb{P}_\pi[A = a|S = s]$.

If the task is supposed to never end (e.g. an inverted pendulum equiped with a motor and trying to balance itself indefinitely), it is called a *continuing task*. If the task is supposed to end when the agent reaches a terminal state S_T, it is called an *episodic task* and each trial of the agent evolving in the environment is called an *episode*.

3.2.1.2 The environment

A MDP $\mathcal{M}$ is defined as $\mathcal{M} =< \mathcal{S}, \mathcal{A}, P, R, \gamma >$, where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, P is transition probability function, R is the reward function and γ is the discount factor (defined later).
The environment is defined by a model composed of the reward function R and the transition probability function P.
When the agent is in a given state S_t and performs the action A_t, allowing it to move towards another state S_{t+1} and to receive a reward R_{t+1}, we say that the *transition* $(S_t, A_t, S_{t+1}, R_{t+1})$

has been carried out. The transition probability function P encapsulates all the probabilities of the agent moving from a state S to a state S' after choosing the action A:

$$P(s', r|s, a) = \mathbb{P}[S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a] \tag{3.13}$$

In a MDP, all the states follow the Markov property, meaning that the future states only depend on the current state, and not the previous states visited by the agent:

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, \ldots, S_t] \tag{3.14}$$

The model of the environment can be known by the agent, completely unknown or partially unknown, depending on the task. As shown on Figure 3.4, this leads to a first distinction between RL algorithms:

- **Model-based algorithms:** the model is initially known by the agent, or the model is not known and has to be learned during the learning process. Once the model is completely known, it is used by the agent for planning all its actions.
- **Model-free algorithms:** The model of the environment is not needed by the algorithm and is not used nor learned.

3.2.1.3 The return and the value functions

As mentioned previously, the goal of the agent is to maximize the total sum of discounted rewards, called the *return* G_t (sometimes also called the *discounted future reward*):

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3.15}$$

where R_t is the reward received at the time step t and γ is the discount factor belonging to $]0, 1]$ and allowing to control the influence of the future expected rewards, which may be estimated with large uncertainties.

The value functions are used by the agent in order to assess which states and/or actions are the best in order to maximize the return based on the current policy π.
The *state-value function* $V_\pi(s)$ is defined by the expected return computed from a given state s, following a given policy π:

$$V_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{3.16}$$

The *action-value function* $Q_\pi(s, a)$, also called the *Q-value function* or the *Q-function*, is defined by the expected return computed from a given action a taken in a given state s, following a given policy π:

$$Q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{3.17}$$

The definitions of these value functions are linked as follows [320]:

$$V_\pi(s) = \sum_{a\in\mathcal{A}} Q_\pi(s,a)\pi(a|s) \tag{3.18}$$

3.2.1.4 The Bellman equations

The goal of the agent is to find the optimal policy in order to maximize the return.

This optimal policy can be found by first seeking the optimal value functions:

$$V_*(s) = \max_\pi V_\pi(s) \tag{3.19}$$

$$Q_*(s,a) = \max_\pi Q_\pi(s,a) \tag{3.20}$$

The optimal policy will be the policy that achieves the optimal value functions:

$$\pi_* = \arg\max_\pi V_\pi(s) \quad or \quad \pi_* = \arg\max_\pi Q_\pi(s,a) \tag{3.21}$$

During the deployment of the agent, the action selected in any given state s will be the action a which gives the maximum value of the function $Q_*(s,a)$.
The value function can be updated iteratively during the learning process thanks to the Bellman equations [320]:

$$\begin{aligned} V(s) &= \mathbb{E}[G_t|S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots |S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots)|S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma V(S_{t+1})|S_t = s] \end{aligned} \tag{3.22}$$

In the same way, the action-value function gives:

$$\begin{aligned} Q(s,a) &= \mathbb{E}[R_{t+1} + \gamma V(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \mathbb{E}[R_{t+1} + \gamma \mathbb{E}_{a\sim\pi} Q(S_{t+1}, a) \mid S_t = s, A_t = a] \end{aligned} \tag{3.23}$$

3.2.1.5 The exploration-exploitation trade-off

A well known issue in Reinforcement Learning is the exploration-exploitation trade-off.
Indeed, the RL algorithm needs to explore the environment by interacting with it in order to discover the action and the state spaces, gathering the information needed by the estimations of value functions and the policy.
In the same way, the agent needs to exploit the already gathered knowledge in order to choose the right actions maximizing the return.

If the agent spends too much time exploring the environment, it will never fulfill the initial task. If it does not explore enough, the estimates of its value functions and its policy will not be accurate enough to choose the right actions maximizing the return, and again the agent will not fulfill the task.

In order to fix this problem, several exploration strategies can be employed: Upper Confidence Bounds [18], Boltzmann exploration [55], Thompson sampling [332], etc.

Epsilon-greedy (ϵ-greedy) is the most common exploration strategy employed in RL. The idea is that we define the probability ϵ of taking a random action in the current state. Thanks to this technique, the agent is able to explore its environment with a probability ϵ and to exploit its current estimates of value functions with a probability (1-ϵ). When the agent is exploiting the information, it simply chooses the action a with the maximum action-value $Q(s, a)$ for the current state s. During exploitation, we say that the agent acts *greedily*.

Moreover, the probability ϵ can be variable. At the beginning of the training process, this probability can be high, since the agent does not know anything about the environment and the task to fulfill, so it needs to explore a lot. After a certain time step or some pre-defined thresholds, the probability ϵ is reduced gradually over time, allowing the agent to switch on an exploitation behaviour, since enough information were previously gathered during the exploration behaviour.

3.2.1.6 Policy-based and value-based

As shown on Figure 3.4, a second distinction can be made among model-free methods:

- Policy-based algorithms: it is the category of algorithms trying to approximate directly the optimal policy, without taking the value functions into account. The REINFORCE algorithm [320] is an example of policy-based algorithms.
- Value-based algorithms: it is the category of algorithms which estimates first the optimal value functions, before using the optimal policy as in equation 3.21. The SARSA and Q-learning algorithms in section C.1 are examples of value-based algorithms.

In the following sections, the most classic Reinforcement Learning algorithms will be presented, followed by more recent algorithms used in the next chapters, which took several ideas from various of these classic RL algorithms.

3.2.2 Temporal-Difference learning

Temporal-Difference (TD) learning [319] is a family of model-free and value-based algorithms. Some of the ideas presented in this section will be reused by the SAC algorithm.

3.2.2.1 TD error

These value-based methods seek to estimate the value functions [336]. They update these estimates towards an estimated return called the *TD target*.

For the state-value function $V(S_t)$, the TD target is defined by the expression $R_{t+1} + \gamma V(S_{t+1})$, inspired by the Bellman equations from 3.22. This target is evaluated after a new transition $(S_t, A_t, S_{t+1}, R_{t+1})$ has been carried out, thanks to the information it brings to the agent. $V(S_t)$ is then updated towards this TD target, by applying a weighted sum using a learning rate α, the same hyperparameter as in section 3.1:

$$\begin{aligned} V(S_t) &\leftarrow (1-\alpha)V(S_t) + \alpha G_t \\ V(S_t) &\leftarrow V(S_t) + \alpha(G_t - V(S_t)) \\ V(S_t) &\leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \end{aligned} \tag{3.24}$$

The learning rate α allows to control the importance of the update applied to $V(S_t)$. The term $(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$ behind the learning rate is precisely what is called the TD error, since it is the difference between the previous estimate of the state-value function $V(S_t)$ and the new TD target $R_{t+1} + \gamma V(S_{t+1})$ evaluated after the last transition.

Similarly, we have the following update for the action-value function $Q(S_t, A_t)$:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)) \tag{3.25}$$

3.2.2.2 On-policy and off-policy

Depending on the way the TD errors or the TD targets are computed, an algorithm can be qualified as being *on-policy* or *off-policy* in the literature:

- **On-policy:** if the policy used to compute the TD targets (called the *target policy*) and the policy used by the agent to explore the environment (called the *exploration policy*) are the same, then the algorithm is on-policy.

- **Off-policy:** if the policy used to compute the TD target (called the *target policy*) and the policy used by the agent to explore the environment (called the *exploration policy*) are different, then the algorithm is off-policy.

3.2.2.3 Deep Q-Network

The Deep Q-Network (DQN) algorithm [229][230] is a major improvement in the history of Reinforcement Learning. It is the first algorithm to take concepts from Deep Learning algorithms and to use them in Reinforcement Learning approaches, creating the subfield of Deep Reinforcement Learning (Deep RL). It is the evolution of the Q-value algorithm (presented in appendix C.1.2).

The major drawback of the Q-learning algorithm is that the estimated Q-values are stored in tables representing all the possible pairs of states and actions. When the dimension of either the state space or the action space increases, the problem becomes quickly intractable.

The main idea behind DQN is to fix this issue by using a neural network as a function approximator of the Q-value function. The Q-value function is then labeled $Q(s, a|\boldsymbol{\theta})$, with $\boldsymbol{\theta}$ being a vector containing all parameters of the neural network. These parameters are the weights and the bias of the neural network, as defined in section 3.1.1.

Moreover, the Q-learning can also suffer from instability and divergence during the learning process. DQN fixes also these issues by implementing two additional ideas:

- **Experience replay:** This mechanism [82] allows the transitions $\mathcal{T}_t = (S_t, A_t, S_{t+1}, R_{t+1})$ experienced by the agent to be store in a *replay memory* (also called *replay buffer*) $D = \{\mathcal{T}_1, \ldots, \mathcal{T}_t\}$. The maximum length of D is a hyperparameter needing to be tuned and this buffer will be able to contain transitions taken from different episodes.
  Random transitions are then sampled from this replay memory D and are to compute the updates of the Q-values estimates. These stored past transitions are used during the learning process in the place of the transitions being currently experienced by the agent, meaning that DQN is an off-policy algorithm: the actions found in these past transitions were selected by a past policy and not by the current exploration policy.
  This trick improves the sample efficiency of the algorithm (which means that less samples are required by the agent in order to converge toward an optimal solution). It also allows to remove any correlation between the sequence of observations or samples used to learn the Q-value function, and to smooth over changes found in the data distribution (the sudden changes appearing in the successive observations become less influential).

- **Periodically updated target:** The Q-values are updated in order to move towards the TD target. In DQN, these TD targets are periodically updated, instead of being updated

at each time step like in Q-learning.
Another neural network of parameters $\boldsymbol{\theta'}$ is created by copying the initial parameters $\boldsymbol{\theta}$. This neural network models the *target Q-value function* $Q'(s, a|\boldsymbol{\theta'})$ and its parameters are kept unchanged for C time steps. Every C time steps, the parameters $\boldsymbol{\theta'}$ are updated by copying the parameters $\boldsymbol{\theta}$ and this process is repeated until the end of the training. This neural network will be used to compute the TD targets found in the update equation of the Q-values.
This mechanism allows to overcome the short-term oscillations, making the learning process more stable.

As for the neural networks described in section 3.1, the parameters $\boldsymbol{\theta}$ are updated using the gradient descent algorithm in order to minimize a cost function. Here, this cost function is defined using the TD error:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(s_i,a_i,r_i,s_{i+1})\sim U(D)}\Big[\big(r + \gamma \max_{a_{i+1}} Q'(s_{i+1}, a_{i+1}|\boldsymbol{\theta'}) - Q(s, a|\boldsymbol{\theta})\big)^2\Big] \tag{3.26}$$

where $U(D)$ is a uniform distribution over the replay memory D, and $\boldsymbol{\theta'}$ is the parameters of the fixed target neural network. This cost function includes a formulation TD error similar to equation 3.25. The backpropagation allowing to update the parameters $\boldsymbol{\theta}$ is carried out over samples taken from the replay memory D and uses this new cost function.

3.2.3 Policy Gradient

Policy Gradient (PG) [321] is another family of Reinforcement Learning approaches. Policy gradient algorithms continue to emerge and are among the most recent advances of the machine learning field. There are a large amount of different algorithms, and the most important ones will be presented in this subsection.
While TD learning approaches were value-based, PG methods are policy-based: the policy is directly optimized in order to maximize the return G. As for the reinforcement learning algorithms described in previous sections, PG methods are model-free.

3.2.3.1 Policy gradient theorem

In PG approaches, the policy is modeled by a parametrized function based on the parameters $\boldsymbol{\theta}$ (usually a gaussian distribution). It is noted $\pi_{\boldsymbol{\theta}}(a|s)$ and has to maximize the return (the expected amount of future discounted reward). To do, the agent has to maximize an objective function $J(\boldsymbol{\theta})$ (noted as the cost function in section 3.1) defined as:

$$J(\boldsymbol{\theta}) = \sum_{s\in\mathcal{S}} d^{\pi}(s)V^{\pi}(s) = \sum_{s\in\mathcal{S}} d^{\pi}(s) \sum_{a\in\mathcal{A}} \pi_{\boldsymbol{\theta}}(a|s)Q^{\pi}(s, a) \tag{3.27}$$

where $d^\pi(s)$ is the stationary distribution of Markov chain for $\pi_{\boldsymbol{\theta}}$ (the parameters $\boldsymbol{\theta}$ have not be written for simplicity). It is defined as:

$$d^\pi(s) = \lim_{t \to \infty} P(s_t = s | s_0, \pi_{\boldsymbol{\theta}}) \tag{3.28}$$

and represents the probability of reaching the state s given an infinity amount of time, starting from the initial state s_0 and following the policy $\pi_{\boldsymbol{\theta}}$.

Since the policy is now parametrized, the parameters $\boldsymbol{\theta}$ need to be updated in order to maximize the return $J(\boldsymbol{\theta})$. The *gradient ascent* is used to improve the policy:

$$\boldsymbol{\theta} \;=\; \boldsymbol{\theta} \;+\; \alpha \; \frac{\partial \, J(\boldsymbol{\theta})}{\partial \, \boldsymbol{\theta}} \tag{3.29}$$

with α being the learning rate (as in section 3.1).

From now, the gradient of the return with respect to the policy parameters is noted $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ for simplicity.
This gradient can be computed thanks to the *policy gradient theorem* [321]:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}} \sum_{s \in \mathcal{S}} d^\pi(s) \sum_{a \in \mathcal{A}} Q^\pi(s,a) \pi_{\boldsymbol{\theta}}(a|s) \\ &\propto \sum_{s \in \mathcal{S}} d^\pi(s) \sum_{a \in \mathcal{A}} Q^\pi(s,a) \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s) \end{aligned} \tag{3.30}$$

A complete proof of the policy gradient theorem can be found in [320].
The gradient can then be further developed:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &\propto \sum_{s \in \mathcal{S}} d^\pi(s) \sum_{a \in \mathcal{A}} Q^\pi(s,a) \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s) \\ &= \sum_{s \in \mathcal{S}} d^\pi(s) \sum_{a \in \mathcal{A}} \pi_{\boldsymbol{\theta}}(a|s) Q^\pi(s,a) \frac{\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s)}{\pi_{\boldsymbol{\theta}}(a|s)} \\ &= \mathbb{E}_\pi [Q^\pi(s,a) \nabla_{\boldsymbol{\theta}} \ln \pi_{\boldsymbol{\theta}}(a|s)] \end{aligned} \tag{3.31}$$

where $\mathbb{E}_\pi$ refers to $\mathbb{E}_{s \sim d_\pi, a \sim \pi_{\boldsymbol{\theta}}}$ when both state and action distributions follow the policy $\pi_{\boldsymbol{\theta}}$ (in the case of an on-policy algorithm, where the target and the exploration policies are the same).

3.2.3.2 Actor-Critic architecture

Besides learning the policy in order to maximize the return, PG methods can also learn the value function: either the value function $V(s)$, or the action-value function $Q(s,a)$, or both

of them in some algorithms. While in TD learning methods the value function gives directly the policy, here it helps indirectly to learn the policy, since the value functions are found in the definition of the objective function, equation 3.27, and in the policy gradient theorem, equation 3.31.
The value functions are also modelled with a function parametrized by a set of parameters $\boldsymbol{w}$.

The *Actor-Critic architecture* is composed of two models:

- **The critic:** it updates the model of the value function $V_{\boldsymbol{w}}(s)$ or $Q_{\boldsymbol{w}}(s,a)$. The TD error is used in order to update the parameters of the value function.
  The critic must minimize the square of the Temporal-Difference error:

$$\delta_t(\boldsymbol{w}) = (r_t + \gamma Q_{\boldsymbol{w}}(s_{i+1}, a_{i+1}) - Q_{\boldsymbol{w}}(s_t, a_t))^2 \tag{3.32}$$

  A gradient descent is needed in order to update the parameters of the value function thanks to the gradient $\nabla_{\boldsymbol{w}}\delta_t(\boldsymbol{w})$.

- **The actor:** it updates the model of the policy $\pi_{\boldsymbol{\theta}}$, which will be used to choose the actions carried out by the agent.
  The actor must maximize the objective function $J(\boldsymbol{\theta})$ defined in equation 3.27 and the gradient $\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta})$ needs to be computed thanks to the policy gradient theorem. These gradients will be used to update the parameters of the policy thanks to the gradient ascent.
  The policy is updated towards the directions suggested by the critic.

Here is an example of a basic Actor-Critic architecture, with the critic modeling the action-value function $Q_{\boldsymbol{w}}(a|s)$. Two different learning rates α_θ and α_w are used for the update of the parameters of the actor and the critic respectively.

---

1. Random initialization of $\boldsymbol{\theta}$ and $\boldsymbol{w}$.

2. **For** episode $= 1, \ldots, M$ **:**

   (a) Initial observation of the state s_0 given by the environment.
   Sampling of an action $a_0 \sim \pi_{\boldsymbol{\theta}}(a_0|s_0)$.

   (b) **For** $t = 1, \ldots, T$ **:**

      i. Sampling of a reward $r_t \sim R(s_t, a_t)$ and the next state $s_{i+1} \sim P(s_{i+1}|s_t, a_t)$ by the environment.

ii. Sampling of the next action $a_{i+1} \sim \pi_{\boldsymbol{\theta}}(a_{i+1}|s_{i+1})$.

iii. The actor updates the policy parameters $\boldsymbol{\theta}$ using the gradient ascent with the policy gradient theorem:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_{\boldsymbol{\theta}} Q_{\boldsymbol{w}}(s_t, a_t) \nabla_{\boldsymbol{\theta}} \ln \pi_{\boldsymbol{\theta}}(a_t|s_t) \tag{3.33}$$

iv. The critic updates the value function parameters $\boldsymbol{w}$ using the gradient descent with the TD error:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha_{\boldsymbol{w}}(r_t + \gamma Q_{\boldsymbol{w}}(s_{i+1}, a_{i+1}) - Q_{\boldsymbol{w}}(s_t, a_t)) \nabla_{\boldsymbol{w}} Q_{\boldsymbol{w}}(s_t, a_t) \tag{3.34}$$

v. Update of $a_t \leftarrow a_{i+1}$ and $s_t \leftarrow s_{i+1}$

This Actor-Critic architecture is on-policy since the exploration policy and the target policy are the same. In some implementations found in the literature, the critic and the actor can share lower layer parameters of their network and two output heads estimating the policy and value functions.

3.2.3.3 Soft Actor-Critic

The Soft Actor-Critic (SAC) [116] is one of the most recent policy gradient algorithm. It is a mix of two families of PG approaches:

- **The Trust Region Policy Optimization (TRPO) and Proximal Policy Optimization (PPO) approach:** they are on-policy algorithms implementing a stochastic policy, and they are known to be stable, but to have a low sample efficiency.
  They manage to improve the stability of the training by avoiding parameter updates that would change the policy too much at one step.
  TRPO [294] uses the Kullback–Leibler (KL) divergence (detailed later in this section) in order to measure the magnitude of the changes in the probability distribution over actions between the old and the new policies. During the update of the policy parameters, this measure is enforced to be small enough by respecting specific constraints. The maximization of the objective function $J(\boldsymbol{\theta})$ becomes a constrained optimization problem.
  PPO [295] simplifies the complexity of the TRPO computations by using a simpler objective function, while retaining similar performance. Instead of using the KL-divergence, PPO modifies the objective function $J(\boldsymbol{\theta})$ in order to keep the ratio

$$r(\boldsymbol{\theta}) = \frac{\pi_{\boldsymbol{\theta}}(a|s)}{\pi_{\boldsymbol{\theta}_{\text{old}}}(a|s)} \tag{3.35}$$

inside a specific interval $[1-\epsilon, 1+\epsilon]$ (where ϵ is a hyperparameter), by computing an expectation over all the states and actions.

- **The Deep Deterministic Policy Gradient (DDPG) and Twin Delayed Deep Deterministic (TD3):** they are off-policy algorithms implementing experience replay and a deterministic policy, and they are known to be very sample efficient, but to be pretty unstable.
  DDPG [212] is based on an Actor-Critic architecture and combines ideas taken from DPG [306] and DQN [229], in order to learn simultaneously the policy and the Q-value function.
  TD3 [90] improves the DDPG algorithm by adding several ideas:
  - Training two Q-value networks in order to reduce the overestimation of the value function, as in the Double Q-learning [122] and Double DQN [339] algorithms. The TD errors are then computed using the minimum of the two Q-value networks, in order to favor underestimation (as in the Clipped Double Q-learning algorithm [90]). Like DQN, each Q-value network has its own target network allowing to compute the TD errors of their respective cost function.
  - Delaying the update of the two target networks and the policy network, in order to reduce the variance of the estimations. The policy network, the target policy network and target Q-value network are updated at a lower frequency than the Q-value network.
    In the Actor-Critic architectures, policy and value updates are deeply coupled: value estimates diverge through overestimation when the policy is poor, and the policy will become poor if the value estimate is inaccurate. Delaying the updates helps to avoid this problem.
  - Smoothing the target policy by adding a small clipped random noise to the action selected in the TD error by the target policy network. This can be considered as a smoothing regularization strategy, in order to avoid the overfitting of the policy to narrow peaks in the value function.

Like DDPG and TD3, the SAC algorithm is an off-policy algorithm implementing an Actor-Critic architecture. Like TRPO and PPO, it uses a stochastic policy $\pi_{\boldsymbol{\theta}}$. The term *Soft* found in the name of SAC means that this algorithm is entropy-regularized.

In reinforcement learning, the *entropy* is a measure of the predictability of an agent. The more the policy of the agent is certain of which action is the best for getting the highest cumulative reward in a given state, the lower the entropy of the policy will be. In other words, the lower the entropy is, the more deterministic the policy will be. In the case of SAC, the entropy $\mathcal{H}(.)$ is defined as follows:

$$\mathcal{H}(\pi_{\boldsymbol{\theta}}(.|s_t)) = \mathbb{E}_{a\sim\pi_{\boldsymbol{\theta}}(.|s)}[-\log(\pi_{\boldsymbol{\theta}}(a|s))] \tag{3.36}$$

The policy is now trained with the objective to maximize the expected return and the entropy at the same time, leading to the following objective function needed to be maximized:

$$J(\boldsymbol{\theta}) = \sum_{t=0}^{T} \gamma^t \ \mathbb{E}_{(s_t,a_t)\sim\rho_\pi}[r(s_t,a_t) + \alpha\mathcal{H}(\pi_{\boldsymbol{\theta}}(.|s_t))] \tag{3.37}$$

where α is a hyperparameter controlling how important the entropy will be (called the *temperature*), γ is the discount factor defined in previous sections, and ρ_π is the marginal of the state distribution induced by the policy $\pi_{\boldsymbol{\theta}}(a|s)$, as defined in the DPG section. The temperature α must not be confused with the learning rate allowing to update the neural networks (they will be noted differently here, to avoid confusion).

The entropy measure of the policy is incorporated into the reward in order to encourage exploration: the goal is to learn a policy that acts as randomly as possible, while still being able to succeed at the task. This allows to avoid situations in which the agent might fall into a local optimum behaviour. Moreover, maximizing the entropy can help to capture multiple modes of near-optimal strategies: if there exist multiple options that seem to be equally good, the policy should assign each with an equal probability to be chosen. This maximum-entropy policy can also give more robustness to the agent, allowing it to be more robust to abnormal or rare events occurring during the task.
The SAC algorithm is the follow-up of the soft Q-learning algorithm [115], created by the same authors.

The SAC algorithm used neural networks in order to learn three functions:

- The policy $\pi_{\boldsymbol{\theta}}$ modeled by a neural network with the parameters $\boldsymbol{\theta}$.
- The soft Q-value function $Q_{\boldsymbol{w}}$ modeled by a neural network with the parameters $\boldsymbol{w}$, corresponding to the Q-value function derived from the new entropy-regularized reward.
- The soft state-value function $V_{\boldsymbol{\psi}}$ (or sometimes simply called soft value function) modeled by a neural network with the parameters $\boldsymbol{\psi}$, corresponding to the state-value function derived from the new entropy-regularized reward.

Like DQN, DDPG and TD3, these neural networks are trained using the experience replay mechanism. Moreover, while DDPG had two target networks for each of its learnt functions, SAC uses just one target network assigned to the soft value function and modeled with the parameters $\boldsymbol{\psi}'$.

The soft Q-value function and the soft state-value function are defined using the Bellman equations:

$$\begin{aligned} Q_{\boldsymbol{w}}(s_t, a_t) &= r(s_t, a_t) + \gamma \mathbb{E}_{s_{t+1} \sim \rho_\pi(s)}[V_{\boldsymbol{\psi}}(s_{t+1})] \\ \text{where } V_{\boldsymbol{\psi}}(s_t) &= \mathbb{E}_{a_t \sim \pi}[Q_{\boldsymbol{w}}(s_t, a_t) - \alpha \log \pi_{\boldsymbol{\theta}}(a_t|s_t)] \end{aligned} \tag{3.38}$$

where $\rho_\pi(s)$ and $\rho_\pi(s, a)$ is the marginal of the state distribution induced by the policy $\pi_{\boldsymbol{\theta}}(a|s)$. The soft state-value function includes the entropy.

This gives us:

$$Q_{\boldsymbol{w}}(s_t, a_t) = r(s_t, a_t) + \gamma \mathbb{E}_{(s_{t+1}, a_{t+1}) \sim \rho_\pi}[Q_{\boldsymbol{w}}(s_{t+1}, a_{t+1}) - \alpha \log \pi_{\boldsymbol{\theta}}(a_{t+1}|s_{t+1})] \tag{3.39}$$

The soft value function is here used as a baseline for the computation of the soft Q-value function in order to reduce the variance of the estimations made by the algorithm.

The soft state-value function is trained in order to minimize an objective function based on TD errors [116]:

$$\begin{aligned} J_V(\boldsymbol{\psi}) &= \mathbb{E}_{s_t \sim \mathcal{D}}[\frac{1}{2}\big(V_{\boldsymbol{\psi}}(s_t) - \mathbb{E}[Q_{\boldsymbol{w}}(s_t, a_t) - \log \pi_{\boldsymbol{\theta}}(a_t|s_t)]\big)^2] \\ \text{with gradient: } \nabla_{\boldsymbol{\psi}} J_V(\boldsymbol{\psi}) &= \nabla_{\boldsymbol{\psi}} V_{\boldsymbol{\psi}}(s_t)\big(V_{\boldsymbol{\psi}}(s_t) - Q_{\boldsymbol{w}}(s_t, a_t) + \log \pi_{\boldsymbol{\theta}}(a_t|s_t)\big) \end{aligned} \tag{3.40}$$

where $\mathcal{D}$ is the replay buffer.

The soft Q-value function is trained in order to minimize an objective function based on TD errors and the target soft Q-value network:

$$\begin{aligned} J_Q(\boldsymbol{w}) &= \mathbb{E}_{(s_t, a_t) \sim \mathcal{D}}[\frac{1}{2}\big(Q_{\boldsymbol{w}}(s_t, a_t) - (r(s_t, a_t) + \gamma \mathbb{E}_{s_{t+1} \sim \rho_\pi(s)}[V_{\boldsymbol{\psi}'}(s_{t+1})])\big)^2] \\ \text{with gradient: } \nabla_{\boldsymbol{w}} J_Q(\boldsymbol{w}) &= \nabla_{\boldsymbol{w}} Q_{\boldsymbol{w}}(s_t, a_t)\big(Q_{\boldsymbol{w}}(s_t, a_t) - r(s_t, a_t) - \gamma V_{\boldsymbol{\psi}'}(s_{t+1})\big) \end{aligned} \tag{3.41}$$

where $V_{\boldsymbol{\psi}'}$ is the target network of the soft value network. The parameters of this neural network are updated using a moving average on the parameters of the soft Q-value network:

$$\boldsymbol{\psi}' \leftarrow \tau \boldsymbol{\psi} + (1 - \tau)\boldsymbol{\psi}' \qquad \text{with } \tau \ll 1 \tag{3.42}$$

Like TRPO, the SAC algorithm uses the Kullback–Leibler (KL) divergence [187][188] in order to quantify the similarity between the policy before and the policy after the update of its parameters.
The KL divergence measures how one probability distribution p diverges from a second probability distribution q. It is defined as:

$$D_{KL}(p\|q) = \int_x p(x) \log \frac{p(x)}{q(x)} dx = \mathbb{E}_{x\sim\mathbb{X}}\Big[\log \frac{p(x)}{q(x)}\Big] \tag{3.43}$$

D_{KL} achieves the minimum zero when $p(x) = q(x)$ on all the possible values of x.

SAC updates the policy in order to minimize the following KL divergence:

$$\begin{aligned} \pi_{\text{new}} &= \arg\min_{\pi'\in\Pi} D_{\text{KL}}\Big(\pi'(.|s_t)\|\frac{\exp(Q^{\pi_{\text{old}}}(s_t,.))}{Z^{\pi_{\text{old}}}(s_t)}\Big) \\ &= \arg\min_{\pi'\in\Pi} D_{\text{KL}}\big(\pi'(.|s_t)\|\exp(Q^{\pi_{\text{old}}}(s_t,.) - \log Z^{\pi_{\text{old}}}(s_t))\big) \end{aligned} \tag{3.44}$$

where Π is the set of the potential policies allowed to be used as a model, while still being tractable, and $Z^{\pi_{\text{old}}}(s_t)$ is the partition function used to normalize the probability distribution created from the soft Q-values.

The actor minimizes the following objective function in order to update the policy network:

$$\begin{aligned} J_\pi(\boldsymbol{\theta}) &= D_{\text{KL}}\big(\pi_{\boldsymbol{\theta}}(.|s_t)\|\exp(Q_{\boldsymbol{w}}(s_t,.) - \log Z_{\boldsymbol{w}}(s_t))\big) \\ &= \mathbb{E}_{a_t\sim\pi}\Big[-\log\big(\frac{\exp(Q_{\boldsymbol{w}}(s_t,a_t) - \log Z_{\boldsymbol{w}}(s_t))}{\pi_{\boldsymbol{\theta}}(a_t|s_t)}\big)\Big] \\ &= \mathbb{E}_{a_t\sim\pi}[\log\pi_{\boldsymbol{\theta}}(a_t|s_t) - Q_{\boldsymbol{w}}(s_t,a_t) + \log Z_{\boldsymbol{w}}(s_t)] \end{aligned} \tag{3.45}$$

The partition function $Z_{\boldsymbol{w}}(.)$ is usually intractable, but this is not a problem in practice since it will not contribute to the gradient $\nabla_{\boldsymbol{\theta}} J_\pi(\boldsymbol{\theta})$. [116] gives a proof showing that this policy update will guarantee that $Q^{\pi_{\text{new}}}(s_t,a_t) \geq Q^{\pi_{\text{old}}}(s_t,a_t)$.

Finally, all the networks can be updated multiple times in a row, for only one step of environment sampling. The gradient descents used different learning rates α_π, α_Q and α_V for updating the functions $\pi_{\boldsymbol{\theta}}$, $Q_{\boldsymbol{w}}$ and $V_{\boldsymbol{\psi}}$ respectively. These learning rates and the weighting factor τ used for updating the target soft state-value network are hyperparameters needing to be tuned.

A simplified pseudocode of the SAC algorithm is given:

---

1. Random initialization of the parameter vectors $\boldsymbol{\theta}$, $\boldsymbol{w}$, $\boldsymbol{\psi}$.
   Initialization of the target parameters: $\boldsymbol{\psi}' \leftarrow \boldsymbol{\psi}$
2. **for** each iteration **do**

(a) **for** each environment step **do**

i. $a_t \sim \pi_{\boldsymbol{\theta}}(a_t|s_t)$

ii. $s_{t+1} \sim \rho_\pi(s_{t+1}|s_t, a_t)$

iii. $\mathcal{D} \leftarrow \mathcal{D} \cup \{\ (s_t, a_t, r(s_t, a_t), s_{t+1})\ \}$

(b) **for** each gradient update step **do**

i. $\boldsymbol{\psi} \leftarrow \boldsymbol{\psi} - \alpha_V \nabla_{\boldsymbol{\psi}} J_V(\boldsymbol{\psi})$

ii. $\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha_Q \nabla_{\boldsymbol{w}} J_Q(\boldsymbol{w})$

iii. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha_\pi \nabla_{\boldsymbol{\theta}} J_\pi(\boldsymbol{\theta})$

iv. $\boldsymbol{\psi}' \leftarrow \tau\boldsymbol{\psi} + (1-\tau)\boldsymbol{\psi}'$

Three successive versions of the SAC algorithm have been released by their creators, each time trying to add new ideas:

1. The version which has been described in this section.
2. The same implementation as the first one, but using two critics instead of one [116], in order to reduce overestimations of the Q-values like in the TD3 algorithm described above. Two soft Q-value network are trained independantly, and the network giving the minimum value between both of them is taken when computing the gradients $\nabla_{\boldsymbol{\psi}} J_V(\boldsymbol{\psi})$ and $\nabla_{\boldsymbol{\theta}} J_\pi(\boldsymbol{\theta})$.
3. The same implementation as the second one, but the soft value network is removed and the temperature parameter α controlling the entropy measure is automatically adjusted by an additional neural network [117].

In this work, we will be using the first version of the SAC algorithm.

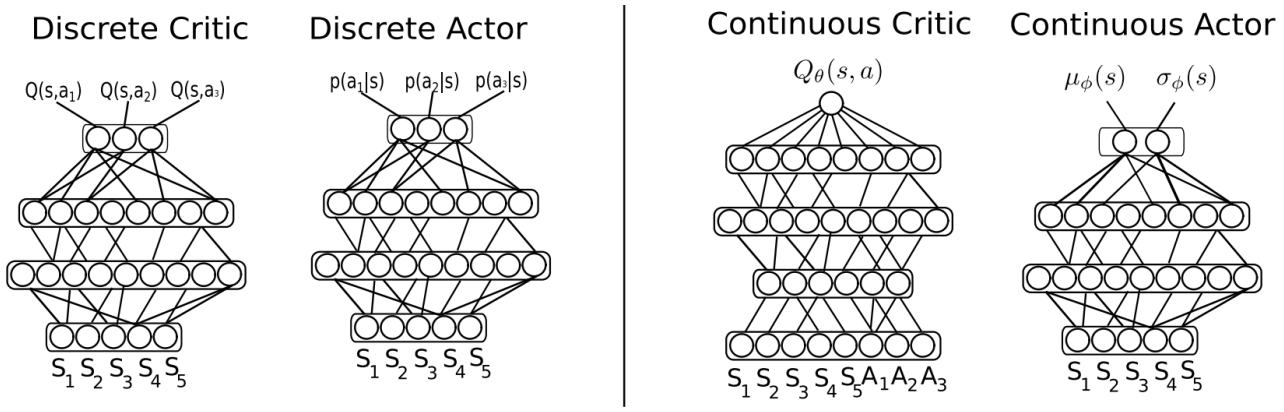


Figure 3.6. Stochastic and deterministic Soft Actor-Critic

As shown on Figure 3.6 [3], SAC can work in both either a discrete action space, or a continuous action space. The soft state-value function V_ψ is not represented here, but its implementation is very similar to the soft Q-value function.

- **Discrete action space:** The critic takes the components of a given state as an input and is outputting the soft Q-values of all the possible actions.
  The actor takes the components of a given state as an input and is outputting the selection probability of all the possible actions.

- **Continuous action space:** The critic takes the components of a given state and a given action as an input and is outputting the soft Q-values computed for these specific state and action.
  The policy is modeled by a *squashed Gaussian distribution* and the action selected by the policy is computed as:

$$\boldsymbol{a} = \tanh(\boldsymbol{n}) \quad \text{where } \boldsymbol{n} \sim \mathcal{N}(\boldsymbol{\mu_\theta}, \boldsymbol{\sigma_\theta}) \tag{3.46}$$

  In this case, the actor takes the components of a given state as an input and is outputting the mean $\boldsymbol{\mu_\theta}$ and the variance $\boldsymbol{\sigma_\theta}$, which can be multidimensional or scalar (depending on number of components of the action).
  The set of all the potential policies Π mentioned earlier is then the family of multivariate Gaussian distributions and is a generic distribution parameterisation choice.

The performance of the SAC is very dependant on the task: sometimes it is better than other algorithms, and sometimes not [5].

[3] Taken from *Soft Actor-Critic*, http://pages.isir.upmc.fr/ sigaud/teach/sac.pdf

Chapter 4

Autonomous Underwater Vehicles and application of machine learning to control theory

In chapter 6, we will apply Soft Actor-Critic, a deep reinforcement learning algorithm, to the control of AUV for a waypoint tracking task. In this chapter, we describe elements of AUV (modelling, GNC systems), as well as a selection of work dealing with the application of deep learning and reinforcement learning to control theory and robotics tasks.

4.1 Autonomous Underwater Vehicles

In this section, we are going to give a brief description of the main design approach of Autonomous Underwater Vehicles (AUVs) [108][149], before describing several examples of control, guidance and navigation algorithms.

The design of an AUV is composed of a variety of sensors estimating the output and the state signals, such as acoustic modems, GPS receivers (used only when the AUV is surfacing), sonars, Doppler Velocity Logs (DVL), Inertial Navigation Systems (INS) also called Inertial Measurement Units (IMU), cameras, etc. The actuators used to move the AUV are mainly the same from one robot to another: they can consist in fins (also called rudders), thrusters (also called propellers), ballasts, etc.
The Figure 4.1 shows an example of AUV (taken from [258]), where only the actuators are detailed. They are composed of only one thruster and four fins (the rudders are just vertical fins). This figure also shows the reference frame (O, X_0, Y_0, Z_0) of the AUV (called the *body-fixed* reference frame) with his different Degrees Of Freedom (DOF): the linear movement along the axes X_0, Y_0 and Z_0 named respectively *surge*, *sway* and the *heave* and the rotational movement along the axes X_0, Y_0 and Z_0 named respectively *roll*, *pitch* and the *yaw*. Roll, pitch

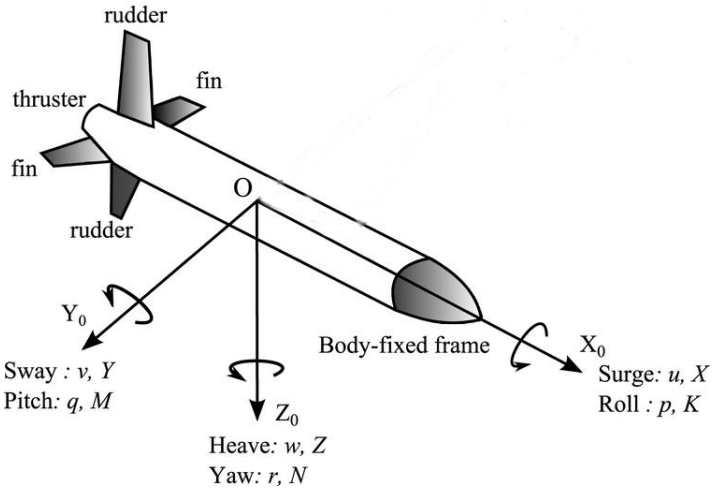


Figure 4.1. Examples of an Autonomous Underwater Vehicle.

and yaw are angles (called the *Euler angles*) and must not be confused with the rotational speeds along the same axes.

There are multiple ways of implementing the components of a GNC system (see section 2.1.2) for an AUV [286], and a mathematical model of the AUV is often needed during the design process.

4.1.1 Fossen's models

The models defined by Thor Inge Fossen in [85] and [86] are very well known among the marine robotics community, and can be applied to a great variety of marine platforms. Fossen's models of AUV will be implemented inside the simulations from the chapter 6.
The formalism used in [86] is based on the subfield of the kinematics, rigid-body kinetics, hydrostatics, seakeeping theory and maneuvering theory. The 6 Degree of Freedom (6DOF) Fossen's model for a general marine vehicle is the following [86]:

$$\begin{cases} \dot{\boldsymbol{\eta}} &= \mathbf{J}_{\boldsymbol{\Theta}}(\boldsymbol{\eta})\boldsymbol{\nu} \\ \mathbf{M}\dot{\boldsymbol{\nu}} + \mathbf{C}(\boldsymbol{\nu})\boldsymbol{\nu} + \mathbf{D}(\boldsymbol{\nu})\boldsymbol{\nu} + \mathbf{g}(\boldsymbol{\eta}) + \mathbf{g}_0 &= \boldsymbol{\tau} + \boldsymbol{\tau}_{\text{wind}} + \boldsymbol{\tau}_{\text{wave}} \end{cases} \tag{4.1}$$

with

$$\begin{cases} \boldsymbol{\eta} &= [x, y, z, \phi, \theta, \psi]^T \\ \boldsymbol{\nu} &= [u, v, w, p, q, r]^T \end{cases} \tag{4.2}$$

where $\boldsymbol{\eta}$ is the vector of the position (x, y, z) (expressed in the *Earth-centered Earth-fixed* reference frame) and the Euler angles (ϕ, θ, ψ), $\boldsymbol{\nu}$ is the vector of the linear velocities (u, v, w)

and the angular velocities (p, q, r), $\mathbf{J}_{\boldsymbol{\Theta}}(\boldsymbol{\eta})$ is a transformation matrix allowing to transform vectors between the *body-fixed* reference frame and the *North-East-Down* coordinate system, $\boldsymbol{\tau}$ is the vector of the control and propulsion forces, $\boldsymbol{\tau}_{\text{wind}}$ and $\boldsymbol{\tau}_{\text{wave}}$ are respectively the vectors of the wind and the wave forces, $\boldsymbol{M}$, $\boldsymbol{C}(\boldsymbol{\nu})$ and $\boldsymbol{D}(\boldsymbol{\nu})$ are respectively the inertia, the Coriolis and the damping matrices, $\mathbf{g}(\boldsymbol{\eta})$ is the vector of the gravitational and buoyancy forces, $\mathbf{g}_0$ is the vector of the static restoring forces and moments due to ballast systems and water tanks. We are not going to further detail how this standard model has been derived.

In the case of an AUV, several physical assumptions [86] allow to simplify the general equations 4.1, leading to the following model:

$$\begin{cases} \dot{\boldsymbol{\eta}} &= \mathbf{J}_{\boldsymbol{\Theta}}(\boldsymbol{\eta})\boldsymbol{\nu} \\ \mathbf{M}\dot{\boldsymbol{\nu}} + \mathbf{C}(\boldsymbol{\nu})\boldsymbol{\nu} + \mathbf{D}(\boldsymbol{\nu})\boldsymbol{\nu} + \mathbf{g}(\boldsymbol{\eta}) &= \boldsymbol{\tau} \end{cases} \tag{4.3}$$

The model of the AUV can be used in order to design model-based GNC components, as well as to build realistic simulations. In the following, we are going to see a selection of AUV controllers taken from the literature of control theory.

4.1.2 Control of AUVs

The literature about the control of AUVs is deep and diverse. We are going to present a selection of works dealing with low-level control tasks applied to AUVs. Several approaches mentioned in the introduction will be cited in this section. Moreover, [273] and [371] are useful surveys about control architectures allowing to give a broad view of the literature. The survey [350] is more specific, since it focuses on the control of underactuated AUVs.

A lot of classical control works make use of the PID controller, the most well-known linear controller. [119] applied a PID controller to the position control and stabilization of an AUV. They used a cascaded position and velocity control approach, based on the inverse kinematic model of a 8 thrusters AUV.
[210] designed an intelligent PID control law, by decomposing a path-following task in separated layers: the 3D path tracking control is decoupled into a planar 2D path tracking task and a distinct depth tracking task. Each layer makes use of its own PID controller.
[107] provides an interesting discussion about the limitations of linear control of AUVs. The authors applied proportional-derivative (PD) controllers to several AUV models linearized for different velocities.
In order to get rid of these limits, PID controllers has been sometimes mixed with other control approaches. For example, [141] compared a fuzzy self-adaptive PID controller and a classic PID controller on an AUV depth control task. [173] also proposed a self-adaptive fuzzy PID controller for the heading and depth control of an AUV. They based their controller on a

nonlinear MIMO model of AUV.

In adaptive control theory, the work [225] used two adaptive proportional-derivative (PD) control laws in order to control an AUV. These two elements only required the model of gravity and the buoyancy regressor matrix for deriving the proposed controller. [225] used adaptive control in order to perform planning with probabilistic state estimation and execution. It allowed to make the AUV adaptive and robust to the dynamic and uncertain conditions of the oceans.

Interesting approaches can also be found in the optimal control theory. [10] was able to compensate actuator failures by using multiple model obtained with optimal control techniques. A switching control mechanism is implemented in order to use the more adequate model. The authors of [67] used optimal control theory in order to perform a path planning task in the horizontal plane. They defined a minimum time problem by constraining the state variables to the places where the control appears linearly.
[45] and [271] both used a Linear–Quadratic Regulator (LQR) in order to control the depth of an AUV. They based their design on a linearized model of the system. The authors of [352] were able to use a LQR controller in order to stabilize the position of an AUV, while decreasing its energy consumption. They optimize the parameters of the controller thanks to a genetic algorithm.
[303] made an AUV fulfill a path-following task thanks to multi-objective Model Predictive Control (MPC) framework. The path tracking is defined as the main objective, while the speed profile is considered as the secondary task. [302] also performed a trajectory tracking task based on a Lyapunov-based MPC. This particular approach allowed to consider explicitly several practical constraints such as the actuator saturation and the thrust allocation. In [251], a docking control is performed thanks to the use of MPC: the model of the AUV is continuously modified during the execution of the task based on the velocity of AUV, the sea currents, and other environment factors. The controller is recalculated at every model change by solving a finite horizon optimal control problems. This type of MPC is called a Receding Horizon Tracking Control (RHTC).

The robust control approaches are very suitable to the control of AUVs, since the marine environment is filled with unknown elements and uncertain parameters can be found inside the model of the AUV. [193] and [287] performed the same robust nonlinear path-following control of an AUV based on Lyapunov theory and back-stepping techniques. The robustness to vehicle parameter uncertainties is addressed by incorporating a hybrid parameter adaptation scheme.
[278] also used a robust control approach with a trajectory tracking task using the second method of Lyapunov. The authors were able to define an uncertain paradigm thanks to the known parameter bounds.
[218] managed to perform the identification of the AUV model using a Kalman filter and

as well as a maximum likelihood estimator. This allowed to obtain an accurate and more representative nominal model, which was then handled by a robust control technique.
[79] used the sliding mode control technique in order to fulfill trajectory tracking task. This allowed the controller to be robust against bounded disturbances. The authors of [284] managed to perform a depth control task based on High Order Sliding Modes (HOSM). They compared their performance with a classical sliding mode algorithm. In [128] a multivariable sliding mode control was implemented for the diving and steering control of an AUV. A combination of the steering, diving, and speed control functions is defined, and the robustness is provided by the variable-structure sliding mode approach.
[242] defined a robust H-infinity based control methodology. The AUV model being formed by highly coupled and non-linear equations, the authors managed to get a reduced order model by subdividing it into smaller subsystems, like depth, steering and speed subsystems. These subsystems are considered to be mutually non-interactive, which allows to facilitates the design of the H-infinity controller.
The authors of [215] compared a H-infinity controller and a sliding-mode controller on a heading and depth control task of an AUV. [205] performed a depth control task using a H-infinity controller coupled with an interval analysis approach. The system inner parameters and the external disturbances are supposed to be time-invariant and bounded in order to apply interval analysis. This method allows to define sets of stability and performance criteria, in any possible time-domain and frequency-domain variations.

The fuzzy logic approach has been employed for the design of advanced AUV controllers by implementing it alone or by coupling it with other techniques. For example, [308] made use of the fuzzy logic for both the low-level and high-level control of AUV. [213] decomposed a 3D path planning problem into two independent 2D planning tasks in the horizontal and vertical planes respectively. The outputs of these two behaviors are fused via a weighted fuzzy logic controller in order to generate the output given to the AUV.
The authors of [114] used the fuzzy logic with a sliding mode controller in order to control an AUV: the fuzzy logic allows to guarantee the stability and the robustness of the control system. [211] mixed the fuzzy logic with backstepping and sliding mode approaches in order to create the adaptive robust control of an underactuated AUV on a path following task.

One of the most easy to implement approach found in the non-linear control theory is the feedback linearization: it is used to transform the complex nonlinear system into a comparatively simple linear system. [56] mixed feedback linearization control with a PD controller. [254] managed to stabilize the roll of an AUV under wave disturbances by using feedback linearization with a H-infinity approach.

The hierarchical control approaches are less often found in the literature of AUV control. The authors of [34] designed a hierarchical control architecture by organizing the controller in

three layers: the mission layer, the task layer and the execution layer. They also implemented a state supervisor and a task coordinator, consisting in two modules able to handle discrete events. [274] defined a hierarchical varying sampling H-infinity control approach for the depth and the pitch control of an AUV. Two different controllers are used, both based on a H-infinity framework with varying sampling intervals.

4.1.3 Guidance and navigation of AUVs

The literature of the guidance and navigation of AUVs is also very extensive. We are going to present a selection of the works relating to these components.

The main guidance or high-level control problems are described in the survey [43]. Several types of mission can be fulfilled by an AUV guidance algorithm, and these missions can be also combined on order to form a multi-objective guidance task.
Path planning tasks consist in generating a trajectory going from an initial point to a final point, while following a given set of constraints. [360] defined an optimal path planning method for coastal environments for AUV, and [373] is a survey focusing on path planning for persistent autonomy of AUVs. These tasks can also make use of intermediate waypoints in order to generate the trajectories, like in [368] and [369].
During obstacle avoidance missions, the system is subject to non-intersection or non-collision position constraints with external elements. [135] and [42] are good examples of AUV guidance algorithms focused on obstacle avoidance.
In marine robotics, station keeping is a very common mission: the robotic platform has to stay inside a limited area, defined around a given point of the environment. [214] describes an algorithm designed for station keeping tasks of AUVs under water wave disturbances. [288] focuses more on a station kereping mission performed with in a power efficient way.

These guidance problems can be solved thanks to different approaches. The line-of-sight (LOS) techniques are one of the most simple implementations of guidance algorithms: the controlled system must follow a target point that is moving in real time along a given trajectory. This trajectory can be either pre-defined or generated automatically thanks to specific constraints. [231] gives an example of a LOS control of an underactuated AUV in the horizontal plane, under the influence of ocean currents. The authors of [365] also proposed a LOS approach for the control of an underactuated AUV subhect to ocean currents, with the addition of input saturations.
The artificial potential fields methods (also called virtual potential field) consist in the definition of a field vector indicating the directions that the system must follow at any point of the environment. The field vector is generated by adding two intermediate field vectors: an attractive field and a repulsive field. The attractive field displays the directions that the

system must follow in order to reach the target points of the environment. The repulsive field displays the directions that the system must follow in order to avoid the dangerous points of the environment or to stay away from the boundaries of the working space. The works [88] and [89] are good examples of local path planning based on an artificial potential fields approach.
The voronoi diagram is a mathematical technique allowing to defined Voronoi cells around specific points or landmarks of the environment. The Voronoi cell associated with a given landmark is composed of all the points which are closer to this particular landmark than to the other landmarks of the environment. The boundaries of these Voronoi cells gives the trajectories which must be followed by the system. [48] used a 3D dynamic Voronoi diagram in order to perform an AUV path-planning task, while [113] performed an obstacle avoidance mission thanks to a Voronoi diagram computed with the sonar data of an AUV.

A lot of navigation tasks can be found in the literature, and the works [227], [260], [208] and [26] (in chronological order) are good surveys describing the major navigation problems as well as the main approaches for solving them.
The most common navigation problem is the localization of the AUV, due to the lack of GPS signals in the marine environment. Both [333] and [169] describes representative methods of acoustic positioning systems.
Another type of widespread missions is the Simultaneous Localization and Mapping (SLAM) tasks: the AUV must localize itself, while updating in real-time the map of an unknown environment. [358] proposed a robust SLAM method applied to an AUV, and [24] designed a SLAM algorithm for AUVs, based on robotic vision and acoustic beacons.

The algorithm allowing to fulfill these navigation missions can be taken from different mathematical subfields. The Extended Kalman Filter (EKF) is one of the most used navigation algorithm, in both the academic domain and the industry. EKF is a probabilistic non-linear estimator able to take into account unknown parameters. Its estimated variables are computed thanks to a linear quadratic estimation based on a set of measurements performed on the observed system. [301] designed an adaptive EKF to the navigation of an AUV, while [9] proposed an EKF in order to provide an AUV navigation robust to the magnetic disturbances of the environment.
Other probabilistic approaches can be applied to AUV navigation problems. For example, [165] used a Bayesian estimation approach based on a distance measurement equipment, as well as several known databases of the environment. The authors of [148] managed to perform the navigation of an AUV under moving ice thanks to a Bayesian estimator. This method allowed to reduce the model errors.
Interval analysis is a set of robust ensemble methods based on constraint propagations and bounding boxes, applied to uncertain variables. These approaches allows to guarantee that the state variables of a dynamical system will remain inside computed regions of the state space. The works [246] and [298] are examples of interval analysis methods applied to the navigation

of AUVs, in order to improve the precision of the state variables estimations.
Uncommon regression techniques can also be found in the literature of AUV navigation algorithms. For example, the authors of [66] designed a hierarchical probabilistic regression in order to perform an AUV navigation based on detection of the fauna and the flora of the marine environment. [81] used an incremental regression method named locally weighted projection regression for the navigation of an AUV.

4.2 Machine learning applied to control theory and robotics

In this section, we will present a selection of works applying machine learning algorithms to control theory and to robotics. The papers related to robotics will mainly focus on AUVs, but other types of platform will be also cited. The machine learning (ML) approaches will be classified in two different subsections: deep learning (DL) and reinforcement learning (RL), the latter also including deep reinforcement learning (DRL). They are the most used ML approaches, even if other techniques can be found in the literature. For example the works [75] and [97] both used a ML control framework based on a genetic programming technique as a search algorithm. This allows to find control laws that are not accessible through linear control theory.

4.2.1 Deep learning applied to control tasks and robotics

First of all, [357] is a good survey about the application of neural networks to control theory.
Neural networks have found early applications in control theory. For example in 1988, [268] designed a multilayered neural network controller, in order to control a general plant model. In 1992, [170] used a neural network in order to control a temperature control system. In 1998, [248] applied a neural network to the control of unknown nonlinear systems. The control signals are directly obtained by minimizing either the instant difference or the cumulative differences between a set point and the output of the neural network.

We can found several works of neural networks coupled with PID controllers. Two approaches have been identified in the literature: either the neural network is working in parallel of the PID controller, or the neural network is replacing the PID controller.
For example, [168] designed a self-tuning PID controller of a flexible micro-actuator using a neural network. The flexible microactuator is made of a bimorph piezoelectric high-polymer material (PVDF). The neural network is trained to reduce the error between the plant output and the reference signal in order to learn the optimal gains of the PID controller. [133] used a similar method by training a neural network in parallel of a MIMO PID controller of an AUV.

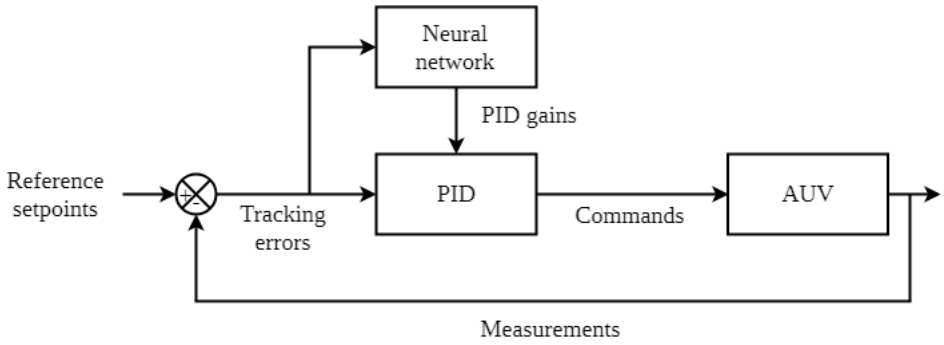


Figure 4.2. A block diagram of a neural network updating a PID controller.

The gains $\boldsymbol{K}_p$, $\boldsymbol{K}_i$ and $\boldsymbol{K}_d$ of the PID controller are updated at each time step by the neural network. A generic representation of this approach is represented on the block diagram of the Figure 4.2.
The approaches replacing the PID controller are based on neural networks reproducing the equations of a PID controller: these are called Proportional–Integral–Derivative Neural Networks (PIDNN). As shown on the Figure 4.3, the neurons of the hidden layer implement the different components of a PID controller, found in the equations 2.7 of the section 2.2. e is the tracking error defined as $e(t) = r(t) - y(t)$, where r is the reference signal that the plant system must follow and y is the measured outputs of the plant system. The weigth matrix of the output layer is composed of the gains $\boldsymbol{K}_p$, $\boldsymbol{K}_i$ and $\boldsymbol{K}_d$, like a PID controller. The command outputted by the PIDNN follows the same equation as in 2.7, except that the gains are updated using the backpropagation technique.
[304] used a PIDNN in order to control general time-delay systems. [202] performed the control of a server fan cooling system thanks based on a PIDNN. In particular, this controller managed to achieve a low power consumption thanks to this approach. The authors of [378] designed a MIMO temperature sensing and control system using PIDNNs. Multiple PIDNN were connected together in order to form a fully-connected PIDNN. This control system allowed to be robust to actuator failures.

The loss functions implemented inside the neural networks replacing or run in parallel of a PID controller are all based on the tracking error e. Several types of expression can be found in the literature for these loss functions:

- The loss function uses only the current tracking error, like in [168] and [378]:

$$J(t) = \frac{1}{2}\,\boldsymbol{e}(k)^2 \tag{4.4}$$

- The loss function is based on the sum of all the squared tracking error obtained from the beginning of the task, like in [202] and [133]:

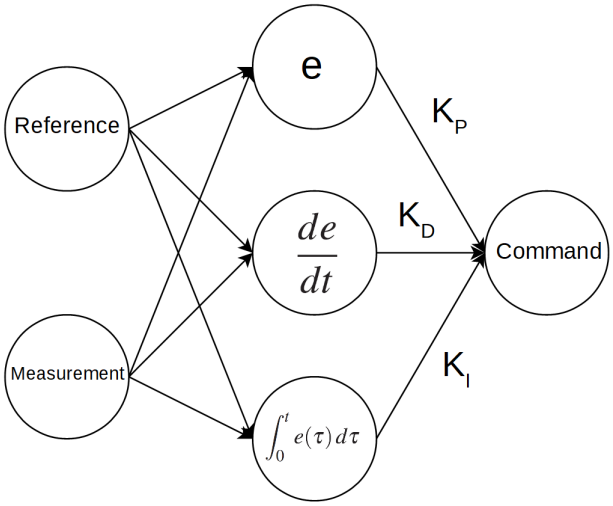


Figure 4.3. Illustration of a PID Neural Network.

$$J(t) = \frac{1}{2} \sum_{k=1}^{t} \boldsymbol{e}(k)^2 \tag{4.5}$$

- The loss function is based on the mean of all the squared tracking error obtained from the beginning of the task, like in [304]:

$$J(t) = \frac{1}{t} \sum_{k=1}^{t} \boldsymbol{e}(k)^2 \tag{4.6}$$

Several great surveys of deep learning applied to robotics can be found in the literature, such as [216], [265], [46] and [166] (in chronological order).
Not a lot of deep learning-based controller of AUVs have been published. The work [63] is one of the best examples: a neural network is used in order to perform an AUV trajectory tracking task using a neural network control approach. Two neural networks (NN) are implemented, based on the AUV model derived in the discrete-time domain. The critic NN is used in order to evaluate the long-time performance of the designed control, while the actor NN is compensating the unknown external disturbances. Even if the authors used the names *actor* and *critic*, this not a RL actor-critic architecture (defined in the section 3.2.3.2).

[317] is an interesting discussion about the limits and the potentials of deep learning approaches for robotics. It mainly deals with the learning challenges, the robotic vision challenges and the reasoning challenges (the inferences or conclusions generated by the processing of any input information). The authors detail the needs for better evaluation metrics, as well as better

simulations for robotic vision. It also talks about the perception, the planning, and the control in robotics, and concludes by saying that the neural networks are generally not performing well when the state of the robot falls outside the training dataset. It also defines the concepts of programming and data as a spectrum allowing to automatically derive deep learning algorithms from reasonable amounts of data and suitable priors.

4.2.2 Reinforcement learning applied to control tasks and robotics

We will present a selection of works aiming to apply RL and DRL to control theory and robotics. We separated the existing literature into three categories: the general surveys and navigation problems together, the low-level control problems and the guidance (also called high-level control) problems. We will show one explicit example of a state vector and a reward function only for the low-level and high-level control problems, since our work will relate more to these subjects.

4.2.2.1 General surveys and navigation problems

Robotics is a breeding ground for RL, since it involves complex task and complex systems. The encountered tasks can be often composed of highly dimensional continuous action and state spaces. [179] is a survey on RL applied to robotics providing a first state of the art. However this article was published in 2013 and the whole DRL subfield is lacking, since the majority of the modern DRL algorithms were not yet published at this time.

A very few papers are covering navigation problems for robotics. [264] applied successfully a plain Q-learning algorithm to an unmanned aerial vehicle (UAV), in order to replace the navigation component of its GNC system. The guidance component consisted in predetermined waypoints, while the control where carried out by a PID controller.
[50] managed to make a DDPG algorithm perform simultaneously the control and the navigation functions of an AUV. The DDPG was tested on a waypoint tracking task.

4.2.2.2 Low-level control problems

Among the low-level control problems, we can find RL approaches applied to PID controller tuning. [140] was an early work trying to tune the gains of a PID controller in order to carry out engine control tasks. The online tuning of the PID was operated by a basic policy-search RL based on Temporal-Difference (TD) errors.
[354] proposed an actor-critic architecture associated with Radial Basis Function (RBF) networks (defined in [234]), in order to create adaptive PID controllers for general non-linear dynamical systems. The same technique was applied to the control of wind turbines systems in [299].

[49] managed to create an adaptive PID controller for terrestrial robots using an original incremental Q-learning approach.

RL papers can also be found for AUV low-level control. [96] is an early trial allowing to control the AUV's thrusters in response to command and sensor inputs. The authors used a Q-learning approach based on a neural network, which was a rare instance of DRL for this time (in 1999).
The RL controller created in [4] is robust to thruster failures. It is based on model-based evolutionary methods: the problem is modeled by a Markov Decision Process (MDP) and the controller is based on a parametrized policy updated by a direct policy search method. The controller is able to operate under-actuated AUVs with fully or partially broken thrusters.
[363] and [51] both implemented the DDPG algorithm in order to control an AUV. The first paper used it in order to create a depth controller allowing track desired depth trajectories, while the second paper allowed the AUV to follow linear velocities and angular velocities reference signals.

Low-level reinforcement learning-based controllers can also be found for other types of systems. [348] is a very early work published in 1965 and experimenting the control of general non-linear dynamical systems using an RL approach. The authors created a model-based controller implementing custom learning schemes.
[127] also managed to control nonlinear dynamical systems thanks to a plain actor-critic architecture based on neural networks. The goal was to control several state variables under actuator constraints.
[364] used a specific TD-learning in order to control turbo-generator systems, while [180] compared the PPO, TRPO and DDPG algorithms to a PID controller, on a attitude control of an UAV.
Finally [118] is a large comparative study Q-leaning associated with neural network. The algorithm is tested on several systems: an AUV, a plane, the magnetic levitation of a steel ball and the heating coil (belonging to the set of Heating, Ventilation, and Air Conditioning (HVAC) problems). These benchmarks allow to evaluated several control theory aspects: effects of nonlinear dynamics, reaction to varying setpoints, long-term dynamic effects, influence of external variables and the evaluation of the precision.

In these problems, the state $\boldsymbol{s}$ of the agent is often composed of a lot of variables in order to correctly follow the reference signals given by the guidance component. For example, the state given to the DDPG in [51] is:

$$\boldsymbol{S}_t = [\boldsymbol{v}, \boldsymbol{\omega}, \dot{\boldsymbol{v}}, \dot{\boldsymbol{\omega}}, \boldsymbol{u}_{t-1}, \boldsymbol{e}_t]^T \tag{4.7}$$

where $\boldsymbol{v}$ is the vector of the linear velocities (v_x, v_y, v_z), $\boldsymbol{\omega}$ is the vector of the angular velocities $(\omega_x, \omega_y, \omega_z)$, $\dot{\boldsymbol{v}}$ is the vector of the linear accelerations, $\dot{\boldsymbol{\omega}}$ is the vector of the angular

accelerations, $\boldsymbol{u}_{t-1}$ of the vector of the commands executed by the AUV thrusters at the previous time step (composed of 6 inputs) and $\boldsymbol{e}_t$ are the tracking errors between the velocities values at time t and fixed time setpoints (the task is here to control the three linear velocities and two of the angular velocities, in order to make them follow several reference signals). This state vector is composed of 23 dimensions.
The reward has also to be specifically designed for each control task. In [51], the reward is defined as follows:

$$r_t \;=\; \lambda_1 \exp\left(-\frac{1}{a^2}(\boldsymbol{x}_t^c - \boldsymbol{x}_{ref}^c)^T \wedge (\boldsymbol{x}_t^c - \boldsymbol{x}_{ref}^c)\right) \;- \lambda_2 \sum_{i=1}^{6} |u_i| \;- \lambda_3 \|\bar{\mathbf{u}}_{t-\tau:t-1} - \mathbf{u}_t\| \qquad (4.8)$$

where $\wedge$ is the cross product operator, λ_1, λ_2 and λ_3 are real-value scalars weighting the three components of the reward, x_t^c is the vector of the controlled variables, x_{ref}^c is the vector of the references values, $\mathbf{u}_t$ is the vector of the commands send to six thrusters of the AUV, u_i is one of the six components of the vector $\mathbf{u}_t$, $\bar{\mathbf{u}}_{t-\tau:t-1}$ is a moving average of the past command vectors based on a slide windows of length τ, and a is a scalar hyperparameter. This reward is composed of three parts:

- The term multiplied by λ_1 which evaluate the square error between the controlled variables and their references.
- The term multiplied by λ_2 aims to limit the magnitude of the commands given to the thrusters.
- The term multiplied by λ_3 allows to penalize a too great variability in the commands values, preventing from generating successive commands differing too largely. This term can be seen as a penalization of commands with too high frequencies signals.

Our work [309] was the first to propose the application of the SAC algorithm to the control of AUVs. [257] is a very recent work (currently in preprint) proposing a design of reward function for an AUV docking task. They tested this reward function formulation by comparing its implementation inside the PPO, TD3 and SAC algorithms, and managed to achieve successful results.

4.2.2.3 Guidance or high-level control problems

We selected several guidance problems relating mainly to AUVs. These approaches are often able to replace both the control and guidance components of the systems, but this is not systematic. [316] implements a classic actor-critic architecture in order to carry out the waypoint tracking and obstacle avoidance tasks of an AUV. [125] is also able to make an AUV fulfill path following and collision avoidance missions, but using a PPO algorithm.

[144] is able to perform a trajectory tracking task of an AUV using the DPG algorithm and Recurrent Neural Networks. The motion control is only done in a 2D horizontal plane. It compares this method with a PID controller and other non-recurrent methods.
[349] is an original work, since it uses the DDPG algorithm in order to plan the trajectories of multiple AUVs. The goal is to estimate a water parameter field inside an under-ice environment.
Finally [262] experiments different modifications of the REINFORCEMENT and the actor-critic algorithms in order to generate the reference signals of the motors of a seven degrees of freedom anthropomorphic arm. The goal is to hit a baseball, and reference signals are followed by PD controllers.

[125] gives a good example of state vector used in guidance problems. It managed to perform path following and collision avoidance tasks by giving to PPO the following vector:

$$\mathbf{S}_t = [\ \boldsymbol{\Theta},\ \boldsymbol{v},\ \boldsymbol{\omega},\ \chi_e,\ \rho_e,\ \boldsymbol{c}_v\ ]^T \tag{4.9}$$

where $\boldsymbol{\Theta} = (\phi, \theta, \psi)$ is the orientation vector of the AUV (the Euler angles), $\boldsymbol{v}$ is the linear velocity vector, $\boldsymbol{\omega}$ is the angular velocity vector (the derivative of $\boldsymbol{\Theta}$), χ_e and ρ_e are respectively the course (or azimuth) error and the elevation error (it corresponds to the error angles indicating the direction of the correct trajectory), $\boldsymbol{c}_v$ is the linear velocity vector of the surrounding ocean currents. This state vector is composed of 14 dimensions.
[316] is performing a waypoint tracking task with obstacle avoidance with the following reward function:

$$r_t = \begin{cases} r_a & \text{if arrive} \\ r_b & \text{if collide} \\ r_c v_x \cos(\nu) & \text{every step} \\ r_d & \text{every step} \end{cases} \tag{4.10}$$

If the AUV reaches the waypoint, it gets the positive reward r_a. If it collides with an obstacle, it receives the negative reward rb. At every time step, it gets the variable reward $r_c v_x \cos(\nu)$: r_c is positive, v_x is the forward speed of the AUV and ν is the angle between the heading of the AUV and the waypoint (in the top-down horizontal plane). The expression $r_c v_x \cos(\nu)$ encourages the AUV to maintain its heading and its speed toward the waypoint. At every time step, the AUV also receives a negative r_d, in order to encourage it to reach the waypoint faster.

Chapter 5

Safe reinforcement learning

When Reinforcement Learning (RL) algorithms are used with physical systems, a lot of constraints and safety concerns appear naturally. The safety must be guaranteed for the system but also for the environment. Safe Reinforcement Learning is the subfield that tackles these problems by mixing notions from RL, optimization and control theory.

In order to give an exhaustive viewpoint of Safe RL, we followed here the very convenient taxonomy defined in [92], while adding some references released after this work and harmonizing specific notations with the notation used during the previous sections. The authors of [92] describe a large number of use cases, involving different risk metrics and safety considerations. They identified two main trends amongs all of these algorithms: the methods that modify the optimization criterion used in reinforcement learning, adding terms alongside of the expectation of the return, while other approaches modify the exploration process itself. This taxonomy is shown on Figure 5.1.

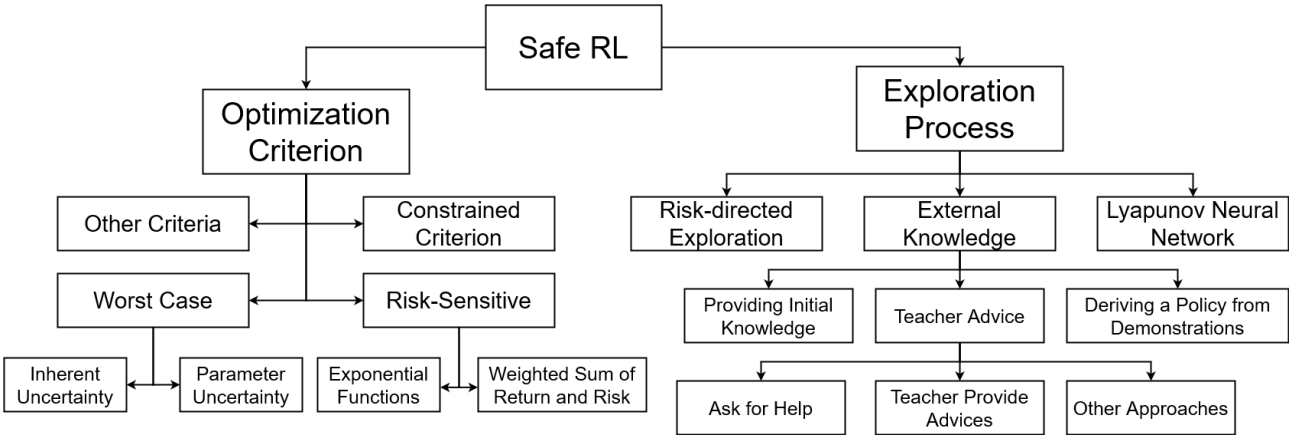


Figure 5.1. A taxanomy of safe Reinforcement Learning approaches.

A few approaches presented in this chapter can be useful for our work (in chapter 6). However, we chose to describe the complete taxonomy of Safe RL, since a lot of these algorithms are based on control theory elements. We believe that these methods can be very inspirational for future works of RL applied to robotics or control theory.

5.1 Adapting the optimization criterion

As described in section 3.2, the primary goal of an agent is to maximize the long term cumulative rewards it gets from the environment throughout an entire episode, at any cost. It is able to learn the optimal policy for a given task by updating some parameters (of the policy, the value function, etc) in order to maximize an objective function composed of the expectation of the return:

$$\max_{\pi \in \Pi} E_\pi(G_0) \; = \; \max_{\pi \in \Pi} E_\pi(\sum_{t=0}^{\infty} \gamma^t R_t) \tag{5.1}$$

where π is the policy of the agent which belongs to the policy set Π, $E_\pi(\ .\ )$ is the expectation with respect to a given policy π, γ is the discount rate and R_t is the reward received by the agent at time t. This objective function is called the *risk-neutral criterion*, since it is totally insensitive to any form of safety guarantees.

A first approach to guarantee the safety of a reinforcement learning algorithm is to modify the risk-neutral criterion in order to take into account some forms of risk. The concept of risk is specific to the subfield of Safe RL and can take various forms, depending on the context.
In the category of the approaches modifying the optimization criterion, the risk is related to the fact that even an optimal policy may perform poorly in some cases due to the variability of the problem, and the fact that the model of the environment is partially known.

5.1.1 Worst-case or minimax criterion

The *worst-case or minimax criterion* is used during the learning process of a task where it is crucial to limit the impact of the worst case scenario, even if it is very unlikely to happen.

The worst-case criterion allows to penalize the variability of the policy. This variability can come from two sources of uncertainty: the *inherent uncertainty* of the stochastic system and the *parameter uncertainty* of the model of the MDP itself.

- The worst-case criterion used under inherent uncertainties [130] corresponds to the case where the environment is stochastic and the agent is not sure to always visit the same

state S_{t+1} every time it starts from the same state S_t and it chooses the same action A_t. The risk-neutral criterion is then modified as follows [129]:

$$\max_{\pi \in \Pi} \min_{w \in \Omega^\pi} E_{\pi,w}(G_0) = \max_{\pi \in \Pi} \min_{w \in \Omega^\pi} E_{\pi,w}(\sum_{t=0}^{\infty} \gamma^t R_t) \tag{5.2}$$

where ω^π is the set of all the trajectories that occur under the policy π and $E_{\pi,w}(\ .\ )$ is the expectation with respect to a given policy π and following a given trajectory w.
With this criterion, the optimal policy corresponds to the policy that has the best worst-case return: the highest return regarding the worst possible trajectory in terms of states visited and actions taken. This is why it is also called the minimax criterion.

The new criterion can be applied to various algorithms. For example, [95] makes use of this criterion to adapt the Q-learning algorithm, becoming the $\hat{\text{Q}}$-learning. In this method the $\hat{\text{Q}}$-values act like a lower bound on the actual Q-values.
It is also proven that this algorithm is overly pessimistic, so it has to be used in critical use cases, where it is imperative to avoid rare occurrences of large negative returns, which can be catastrophic on some systems.

- The worst-case criterion under parameter uncertainties [247] differs slightly from the previous one, since these uncertainties correspond to model errors. Indeed, it is used during tasks where the transition matrix defining the environment of the MDP is not exactly known and belongs to a family of possible models [324]. These uncertainties can be due to estimations of the model from noisy data, estimations from insufficient training examples (or learning samples) or variable data which change during the execution of the policy, which invalidate the estimations coming from them.
The risk-neutral criterion is then modified as follows:

$$\max_{\pi \in \Pi} \min_{p \in P} E_{\pi,p}(G_0) = \max_{\pi \in \Pi} \min_{p \in P} E_{\pi,p}(\sum_{t=0}^{\infty} \gamma^t R_t) \tag{5.3}$$

where p is a transition matrix belonging to the set of all possible models P and $E_{\pi,p}(\ .\ )$ is the expectation with respect to a given policy π and for a given transition matrix p of the MDP.

This is a problem similar to those founded in the robust control theory [380], where the algorithms are designed for rejecting external disturbances and for being robust to model uncertainties [120].

5.1.2 Risk-sensitive criterion

The principle of risk-sensitive criteria is that it allows to find a trade-off between getting large cumulative rewards and avoiding risky situations. Indeed the criterion includes a scalar parameter β allowing the decision maker to decide to what extent the policy will be aware of the notion of risk.
If $\beta > 0$ a risk aversion behaviour is promoted, and if $\beta < 0$ a risk-seeking behaviour will be encouraged. $\beta = 0$ implies a risk neutrality behaviour.

This criterion is mainly modeled using two different approaches:

- Using exponential functions, as in [139][39]:

$$\max_{\pi \in \Pi} \beta^{-1} \log E_\pi \left(\exp^{\beta G_0} \right) \; = \; \max_{\pi \in \Pi} \beta^{-1} \log E_\pi \left(\exp^{\beta \sum_{t=0}^{\infty} \gamma^t R_t} \right) \tag{5.4}$$

  Using a Taylor expansion of the exponential and logarithm functions, the criterion can be expanded as:

$$\max_{\pi \in \Pi} \beta^{-1} \log E_\pi \left(\exp^{\beta G_0} \right) \; = \; \max_{\pi \in \Pi} E_\pi(G_0) \; + \; \frac{\beta}{2} Var(G_0) \; + \; \mathcal{O}(\beta^2) \tag{5.5}$$

  where $Var(G_0)$ corresponds to the variance of the return.

  Thanks to these expansions, we see that the risk metric is here associated with the variance of the return, since the parameter β regulating the risk awareness of the agent is in front of the term $Var(G_0)$. Indeed, a high variance in the return implies instability in the received rewards and so more risk, since the return can be largely positive at a given time step, before being largely negative (and so catastrophic) a few time steps latter.
  However, the use of exponential functions is not suited for problems where a policy with a small variance can produce a large risk, since the small variance will not alert this criterion of the actual risk occurring.

- Using a weighted sum of the return and the risk [289]:

$$\max_{\pi \in \Pi} \left(E_\pi(G_0) \; - \; \beta \omega \right) \tag{5.6}$$

  with ω being the risk metric.
  The use of a weigthed sum allows to directly balance the priority over maximizing the return or the risk awareness.

  This risk metric can be replaced by various terms: the variance of the return [111], the temporal difference errors that occur during learning [226], or even the probability of the

agent reaching an error state, following a given policy and starting from a given state [100].
Each possibility induces a different behaviour and this choice is very task-dependent. [226] and [100] proved that the risk-sensitive behaviour is induced at the cost of indirectly transforming the action-values $Q(S_t, A_t)$. It means that the estimation of the long term utility of the actions is loose and that the agent is able to detect long term risk situations but not risk situations in the early steps of the learning process. The policy may be overly pessimistic.

A trade-off must then be found between the risk-awareness and the maximization of the cumulative rewards, depending of the critical aspect of the task.

5.1.3 Constrained criterion

Constrained criteria [232][164] do not consist in a real modification of the optimization criteria. The classic expected return is used (the risk-neutral criterion) but optimization constraints are added, in order to keep other expected measures within specific bounds.
These constraints leads to the definition of sets of allowable or safe policies inside the whole policy space.

Some constraints can be applied to the expectation of the return [99], in order to keeping it superior to a minimal value: $\mathbb{E}(G_0) \geq k$.
The variance of the return [323] can also be constrained to stay inferior to a minimal value: $\text{VAR}(G_0) \leq k$.
Other approaches are based on the principle of *ergodic MDPs* [147], which guarantee that any state is reachable from any other state by following a suitable policy. The space of safe policies is restricted to the policies preserving the ergodicity with a specific probability called the *safety level*, which has to be tuned.
More exotic constraints can also be found. For example, [3] developed a constrained RL algorithm in a tax collection system and used legal, business and resource constraints.

The drawbacks of these methods is that many of these problems are computationally intractable, which adds difficulties to the formulation of RL algorithms. Moreover it can be hard to choose the correct parameters that will have to be constrained, and not all the constraint types can be applied to all the domains.

5.1.4 Other optimization criteria

The area of financial engineering [244][105] has been at the origin of several very specific optimization criteria (not detailed in this section) such has the *Sharpe ratio* also called the *Value-at-Risk (VaR)* [217][323] or the *density of the return* [238][239].

5.2 Changing the exploration process

In the usual exploration process, the agent starts the training phase by discovering its environment and learning the task from scratch, by selecting random actions. This behavior can make the agent fall in an undesirable or unsafe state, which could harm the environment or the physical system. Moreover, exploring the state and action spaces can take a lot of training time before finding interesting or rewarding pairs of states and actions.
The other main trend of safe reinforcement learning is to change the exploration process, while keeping the classic risk-neutral optimization criterion.

5.2.1 Using external knowledge

The external knowledge corresponds to the knowledge that is not gathered by the interactions of the agent with the environment.
It "informs" the agent about the unsafe regions of the state space, allowing it to know the undesirable or harmful states without having to visit them to discover their unsafe nature. Another advantage is that this process saves a lot of time and is able to accelerate the training phase, which always beneficial for the reinforcement learning workflow.

5.2.1.1 Providing initial knowledge

Prior knowledge of the problem can be provided in order to bootstrap the training phase [310][221]. This knowledge is first gathered and formatted by a human teacher, leading to a finite set of demonstrations. Then this set is used by a regression algorithm in order to produce a partial value function which will guide the exploration during the beginning of the training phase.
Following this initialization, the system can switch to a Boltzmann or fully greedy exploration based on the values predicted in the initial training phase.
In this way, the learning algorithm is exposed to the most relevant regions of the state and action spaces from the earliest steps of the learning process, thereby eliminating the time needed in a random exploration for the discovery of these regions.

Transfer Learning (TL) [329][330] can also be used as another way to provide initial knowledge to the agent: the experience gained in learning to perform one task can help to improve learning performance in a related, but different task.

However, a bias can be introduced by this initial knowledge, which may produce sub-optimal policies. The exploration process following the initial training phase can also make the agent visiting catastrophic or unsafe states. Finally, it can be difficult to initialize some complex RL structures.

5.2.1.2 Deriving a policy from a finite set of demonstrations

This approach of Safe Reinforcement Learning is based on techniques of the two subfields of *Learning from Demonstration (LfD)* [290][8] and *apprenticeship learning* [2][326].

It is quite similar to the methods of the previous section, since a teacher also provides demonstrations of the task, recording generated state-action trajectories, in order to produce a model learnt from these demonstrations. But here this model is used to directly derive a near-optimal policy, instead of deriving a value function.

However, the learner performance is heavily limited by the quality of the teacher's demonstrations, and the agent do not know how to act when it encounters a state for which no demonstration exists. The authors of [92] also explained that these methods have mainly been applied to model-based RL algorithms.

5.2.1.3 Using teacher advice

Instead of initializing the training phase as it was the case for the two previous sections, the advice given by the teacher are used by the agent throughout the entire exploration of state and action spaces.
The teacher is available for the agent at any time of the training phase and can be a human or another controller. The teacher advice consist in giving actions that will maximize the optimization criterion chosen for the task.

Depending on the needs of the task, the interactions between the learner agent and the teacher can be initiated by the agent [59][101] (in *ask for help* approaches), by the teacher [60][331] (in *teacher provides advices* approaches), or by neither of them [277][185] in other specific approaches.

5.2.2 Risk-directed exploration

The risk-directed exploration methods use a risk metric in order to adjust the probabilities of the actions selected by the agent.

For example, the risk metric can be based on the notion of *controllability* [98]: the more a state or a state-action pair leads to a lot of variability in the Temporal-Difference (TD) error signal, the less it is controllable.
This notion of controllability do not have to be mistaken for the property of controllability found in the control theory [131]. In the control theory, a dynamical system completely

controllable is a system that can reach any of its intern states.

In [197], the measurement of the risk for a particular action in a given state is the weighted sum of the entropy and normalized expected return of that action. The notion of entropy is the same as the one defined in the implementation of the Soft Actor-Critic algorithm found the section 3.2.3.

5.2.3 The Lyapunov Neural Network

The Lyapunov Neural Network (LNN) [272] are special neural networks trained in order to estimate the Lyapunov functions (defined in the section 2.3.4).
It allows to certify the safety of a closed-loop dynamical system (see section 2.1.2).

The weights parameters of the neural network are constrained in order to follow a specific structure, and the loss function, the ground-truth labels and the backpropagation are redefined using mathematical justifications. These modifications allows to have an estimate of the Lyapunov function at any time, given by the output of the modified neural network, and a region of attraction can be defined in the state space of the closed-loop dynamical system.

The LNN can be employed in order to constrain the closed-loop dynamical system to stay in the approximated regions of attraction during the exploration process of controllers based on reinforcement learning. Moreover, LNN can be applied to all kinds of controllers, based on machine learning or not. The plant system being controlled can be non-linear, but a mathematical model is needed by the LNN.

The Figure 5.2 (taken from [272]) shows the region of attraction approximated in the state space of an inverted pendulum (defined in the section 2.1.1), around the equilibrium $(0, 0)$. It is compared with regions of attraction approximated using other classical algorithm, and we can see that the regions of attraction found with a LNN (in orange) is the larger, allowing to give more freedom to the agent during the exploration process while guaranteeing its safety. The results are displayed on a phase portrait.

The Lyapunov Neural Network is the follow-up of other works from the same authors [31][30][32].

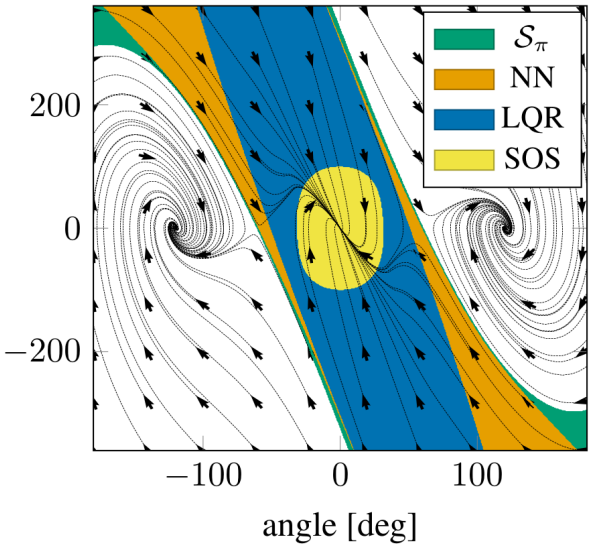


Figure 5.2. A comparison of the region of attraction of an inverted pendulum estimated by different algorithms.

Chapter 6

Proposals for the development of deep reinforcement learning-based controllers for AUV

We will describe in this chapter all the results we obtained thanks to the use of methods introduced in the previous chapters, as well as how we adapted them for the control of an Autonomous Underwater Vehicle (AUV). We only worked with simulations and not with real robotics platform, but we kept in mind that all the developed solutions could be embedded on real AUVs in the future. We will first explain how our simulations are structured, before describing the AUV model we chose and the control task we want to fulfill. The goal of these simulations is to compare the performance of the PID controller and the SAC algorithm on waypoint tracking tasks.
After detailing the implementation of the PID and the SAC and defining the performance metrics we used, we will describe all the trials we made in chronological order. We will start with our first attempt of making the Soft Actor-Critic (SAC) fulfill the control task (these results were published in [309]). We then performed a sensitivity analysis in order to study how the number of state variables provided to the SAC algorithm can affect its performance. We used a simpler waypoint tracking task for this sensitivity analysis. We also implemented advanced training techniques in order to see if it can improve the SAC performance on a harder waypoint tracking task. We will finally propose a training methodology, as well as a discussion about the integration of guidance functions inside our end-to-end controller.

6.1 The setup of the simulations and the control solutions

In this section, we will describe the needs which led to the simulation tools we chose, before explaining how these tools interact with one another. Simulation is an important research and development tool that can be used to test newly devised control algorithms on a vehicle. Simulation enables algorithms and control schemes to be evaluated in a virtual environment thus reducing potential risks associated with real-world experimentation. It facilitates the transfer of the controller from simulation to the real world.
We will also describes the AUV model, the control task, the two different controllers as well as the metrics chose to measure their respective performance.

6.1.1 Simulation tools

Deep reinforcement learning algorithms need to be trained on a realistic simulation before being deployed on a real robotic platform. Indeed, simulation plays an important role in the development, testing and evaluation of new robotic applications, reducing implementation time, cost and risk.

The simulated environment needs to be representative of the ground truth and has to give rewards and appropriate state observations to the deep reinforcement learning agent.
As explained in the introduction, the marine environment is very hostile and is composed of a lot of unexpected events and external disturbances. The chosen tools needs to simulate these factors.

6.1.1.1 Choosing the right simulation tools

Training a deep reinforcement learning algorithm with a realistic simulation in order to fulfill an Autonomous Underwater Vehicle (AUV) control task requires multiple needs.

- **Programming language:** A *programming language* is required in order to implement the SAC algorithm, the PID controller and to manage the whole simulation and its results. The *Python* programming language [219] is a natural choice for this work, since it is one of the most used language in data science [102].
  The Python library *NumPy* [252] was used in order to manage some array data and to perform scientific computations on the Central Processing Unit (CPU) of the PC.
  We also made use of the Python library *Matplotlib* [143] in order to print the evaluation of the loss functions of the neural networks of the SAC in real time (during the training process).

- **Machine learning framework:** A *machine learning framework* is needed in order to simplify the implementation of the SAC algorithm. ML frameworks are libraries allowing to carry out neural network computations on the Graphics Processing Unit (GPU) of a computer. The GPU performs parallel computing and helps to reduce the computation time needed by the execution of a given program. Since the neural network computations are highly parallelizable, the usage of the GPU allows to speed up the learning process.
  The most well-known ML frameworks available for the Python programming language are:

  - *Tensorflow* [1] and *Keras* [102], two libraries which are often used together. Tensorflow is used for low-level implementations of custom neural network architectures, while Keras is used for high-level implementations of state-of-the-art layers of neural networks. These two libraries are mainly popular in the industry domain since they are one of the oldest ML frameworks and their deployment are very time-tested. They also have one of the biggest community of users.
  - *Pytorch* [256], the Python version of the *Torch* framework. It is mainly popular in the academic domain, it is more recent than Tensorflow and Keras, and its features are quickly evolving.

  We chose to use the Pytorch framework [256], since it is more popular in the academic domain than in the industry: we want our work to be reusable for other research projects.

- **Robotics middleware:** a *middleware* is a framework allowing to structure an application in distinct components and to manage all the communications made between all these components. It independantly executes all components in parallel.
  In robotics, a middleware is used to design all parts of GNC system (defined in the section 2.1.2) in multiple software components and allows to easily transfer the application from a simulation to a real robotic platform. Indeed, specific components are implemented for the simulation, the low-level features and the high-level algorithms: when the application needs to be embedded in a real robot, only the useful middleware components are kept. Middlewares also offer useful tools for the supervision and the management of the application.
  The most used robotics middleware is the *Robotic Operating System (ROS)* [1] [269], which is available for more and more programming language and is often updated with the aim of being secured and well optimized. It is applied to a wide variety of robots.
  The *Mission Oriented Operating Suite-Interval Programming (MOOS-IvP)* [2] [28] is another robotics middleware specifically designed for marine robotics. The programming paradigm is here slightly different and MOOS-IvP is less often updated and less robust

[1] https://www.ros.org/
[2] https://oceanai.mit.edu/moos-ivp/pmwiki/pmwiki.php

than ROS.
During this work, we chose to use the ROS middleware because it is a more sustainable solution for future applications of our results and because a lot more of documentation and existing packages are available.

- **Robotics simulator:** a simulator is needed in order to train the SAC algorithm with realistic data. There are various simulators designed for robotics available:

  - **General robotics:** let us see several general robotics simulators first. These simulators were not designed for specific sub-domain of robotics and resources can be found for nearly all types of robotic platforms: terrestrial, aerial, robotics arms, etc. They can often accurately and efficiently simulate several robots at the same time and operate them in complex indoor and outdoor environments. One thing to note is that these general simulators are systematically lacking of marine robotics support by default and need to develop additional plugins or packages. This is due to the fact that the marine robotics community is smaller than other robotics communities. The *Gazebo* simulator [3] [181] is among the most popular robotics simulator since it is based on the well-known ROS middleware: Gazebo can be launch from ROS or independently, and a lot of interactions are available between ROS components and Gazebo.
    The *Modular OpenRobots Simulation Engine (MORSE)* simulator [4] [76] is also found in a lot of use cases. It is based on the *Blender Game Engine* and the *Bullet Physics* engine, and iso its graphics rendering can be better than other robotics simulator. It can be used with a lot of different middleware.
    CoppeliaSim [5], formerly known as *V-REP* [275], is robotics simulator based on a distributed control architecture allowing to used components of different middlewares or frameworks in the same application.

  - **Marine robotics:** the following simulators were specifically designed for marine robotics applications: underwater robots, surface vehicles, underwater sensors, etc. There are often (but not systematically) based on one of the previous general robotics simulator and improved it by adding some lacking features. Indeed they are adding the support of hydrodynamic forces, such as waves or ocean currents, as well as some marine robot models and environments.
    The *Unmanned Underwater Vehicle (UUV) Simulator* [6] [222] focuses on underwater robotic platforms. It is composed of a set of Gazebo and ROS packages and both the hydrodynamics and the robots models are based on the Fossen's models (defined in

[3] http://gazebosim.org/
[4] https://morse-simulator.github.io/
[5] https://www.coppeliarobotics.com/
[6] https://github.com/uuvsimulator/uuv_simulator

the section 4.1). It allows to simulate various sensors and actuators, as well as ocean currents. It includes various models of Autonomous Underwater Vehicles (AUV) and Remotely Operated Vehicle (ROV), and offers several ready-to-use underwater environments and control algorithms.
The project *UWSim* [7] [266] is another simulator of underwater vehicles, offering other examples of AUV models and environment.
Virtual RobotX [8] [35] is also a marine robotics simulator based on ROS and Gazebo, but is exclusively focus on Unmanned Surface Vehicles (USV). Its development comes from the simulation-based robot competition designed to complement the physical Maritime RobotX Challenge.
The project *USV SIM* [9] [255] is another simulator focused only on USVs. It is also based on ROS and Gazebo.
MArine Robotics Simulator (MARS) [10] [335] is a *Hardware-in-the-Loop* simulator for multiple AUVs and USVs, meaning that the inputs and the ouptuts of a real hardware can be plugged to a PC executing the MARS simulator and simulating the rest of the hardware and of the environment.
Other projects can be more focused on the underwater sensors, such as the project described in [37] [11]. This class of simulators aims to have higher granularity, meaning that marine phenomena are simulated with greater precision.

For this work, we choose to use the Unmanned Underwater Vehicle (UUV) Simulator. Indeed, the UUV Simulator is based on ROS (which was the middleware we previously chosen) and have a lot of ready-to-use AUVs, underwater environments and controllers. It also makes use of the Fossen's models, which are sufficiently precised for robotics applications: we do not need to simulate precise micro-phenomena such as acoustic propagation.

- **Statistical analysis tool:** a tool allowing to analyse post-simulation results was needed. We chose to use *MATLAB* [12] [233], because of the simplicity it offers for exploring large datasets and generating plots and figures.

Once the tools have been chosen, they have to been integrated all together.

6.1.1.2 The architecture of our simulations

In this section, we are going to describe how we used these simulation tools all together. The Figure 6.1 explains how the application is structured with all these tools, by recalling the main

[7] http://www.irs.uji.es/uwsim/
[8] https://github.com/osrf/vrx
[9] https://github.com/disaster-robotics-proalertas/usv_sim_lsa
[10] https://github.com/iti-luebeck/MARS
[11] https://github.com/hblasins/uwSim
[12] https://www.mathworks.com/matlab

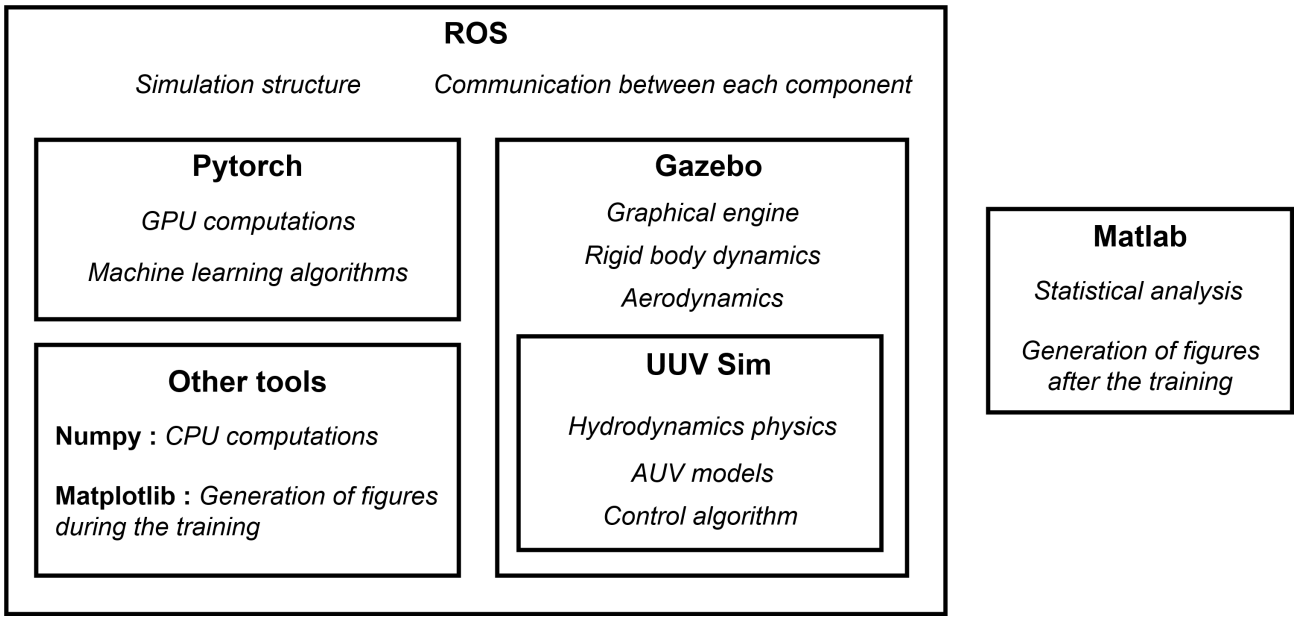


Figure 6.1. Architecture of our marine robotics application.

features they provides and showing how they are included one in another.

Besides Matlab, which is used for post-simulation results analysis, all tools are used inside ROS components. This is why the middleware ROS encapsulates all the other tools in the Figure 6.1, except Matlab. In the ROS terminology, these ROS components are called *Nodes* and are able to communicate between one another thanks to the use of *topics* and *services*. As explained previously, Gazebo is based on a ROS architecture and allows to run independently the physical computations needed by the simulation and control algorithms. Gazebo can be launched without its user interface (called the Gazebo client), allowing to compute the dynamics of the simulation without displaying the rendering. The UUV Simulator is built upon Gazebo and adds new packages. The ROS nodes can interact with the UUV Simulator nodes during the execution of a simulation and can use a variety of features in real time: receiving the state variables of the AUVs, changing the direction and the magnitude of the ocean currents, sending commands to the thrusters, simulating sensor failures, etc.
The Figures 6.2 and 6.3 show two different views of the UUV Simulator:

- The Figure 6.2 shows a view taken from rviz, a tool provides by ROS and allowing to supervise the simulation. The mission displayed on this figure is a path following task performed by the *ECA A9* [13].
- The Figure 6.3 shows a view taken directly from the Gazebo client. The mission displayed

[13] https://github.com/uuvsimulator/eca_a9

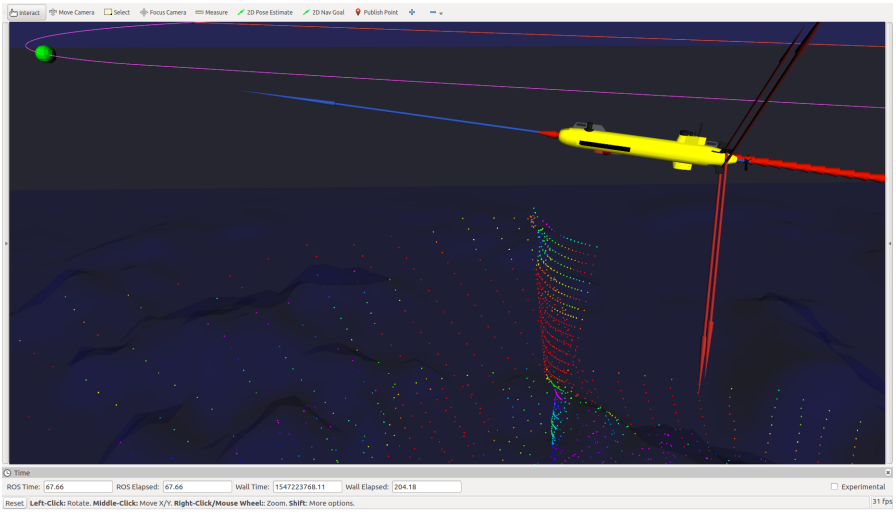


Figure 6.2. The ECA A9 performing a path following task, viewed from rviz.

on this figure is a waypoint tracking task performed by the *RexROV 2*, which will be detailed in the next section.

6.1.2 AUV and task setup

In this section, we will explain how the tools introduced previously are used in order to define the control task and the AUV used to perform it.

6.1.2.1 The autonomous underwater vehicle

The UUV Simulator offers several models of real AUVs and ROVs: the RexROV, the RexROV 2, the ECA A9, the Desistek SAGA ROV and the Light Autonomous Underwater Vehicle (LAUV). These models are based on the Fossen's models defined in the section 4.1.
For this work, we chose to use the RexROV 2 model [14]. The RexROV 2 is usually operated remotely by a human, but here it will be considered as an AUV since it will act autonomously thanks to control algorithms. The dimensions and parameters of the RexROV 2 were derived from model parameters of the SF 30k ROV [29]. As shown on the Figure 6.4, the RexROV 2 is a cube-shaped ROV. It has six degrees of freedom (6DOF), the three possible translations and the three possible rotations, and it is actuated thanks to six thrusters or propellers. These thrusters are placed according to an unconventional layout (see Figure 6.5): one given thruster is able to impact multiple DOFs at the same time, and each basic movement needs a specific combination of thrusters inputs. The RexROV 2 sensors are composed of an Inertial

[14] https://github.com/uuvsimulator/rexrov2

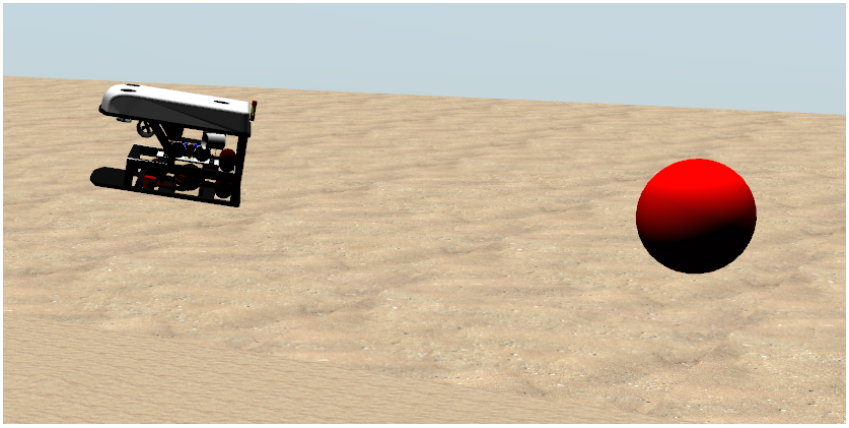


Figure 6.3. The RexROV 2 performing a waypoint tracking task, viewed from the Gazebo client.

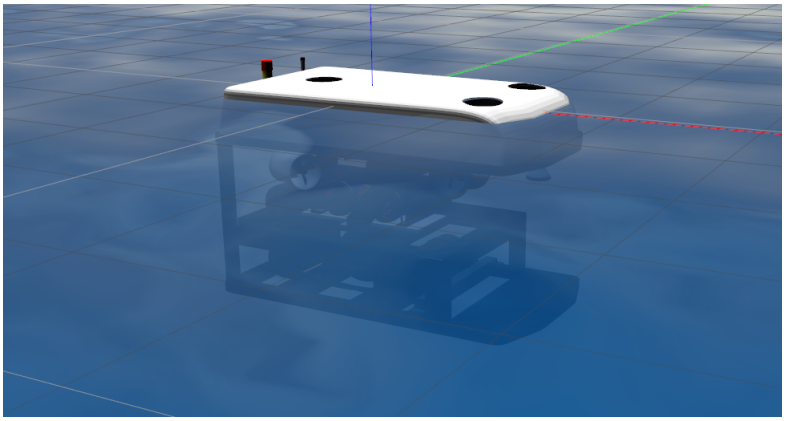


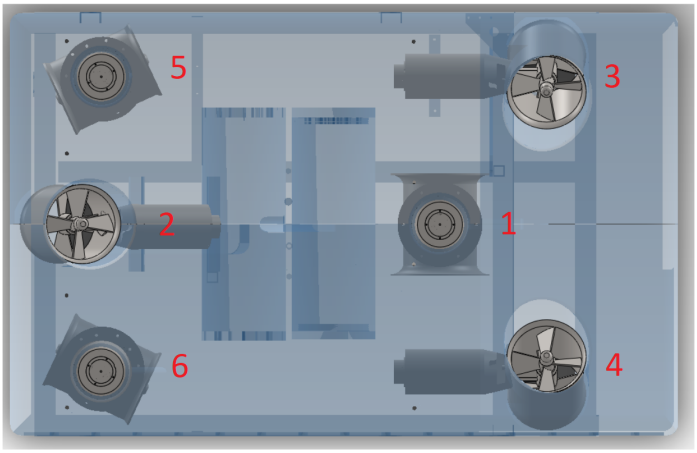
Figure 6.4. The RexROV 2 shown in the Gazebo client.

Measurement Unit (IMU) measuring the linear and angular accelerations and velocities in the AUV frame, a Doppler Velocity Log (DVL) measuring the linear velocities of the AUV relative to the seabed and a pressure sensor measuring the depth of the AUV. It can also have a RGB camera and a GPS sensor (used when the AUV is surfacing) but there were not implemented in this work. This particular AUV was chosen for its shape allowing an ease of controllability.

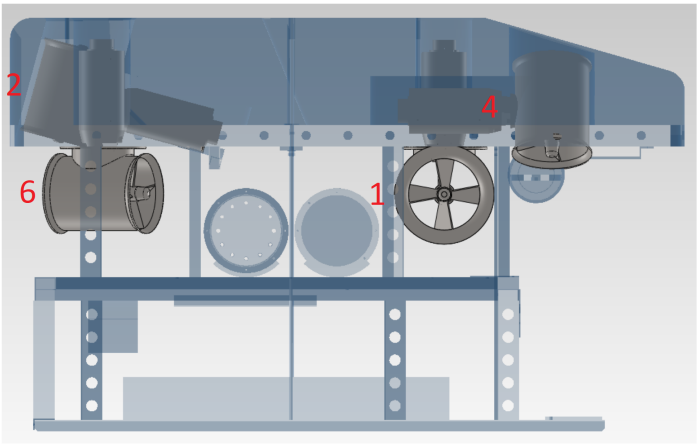
The notation used to describe the RexROV 2 variables will be the following:

$$\begin{cases} \boldsymbol{\eta} &= [x, y, z, \phi, \theta, \psi]^T \\ \boldsymbol{\nu} &= [v_x, v_y, v_z, \omega_x, \omega_y, \omega_z]^T \\ \boldsymbol{u} &= [u_1, u_2, u_3, u_4, u_5, u_6]^T \end{cases} \tag{6.1}$$

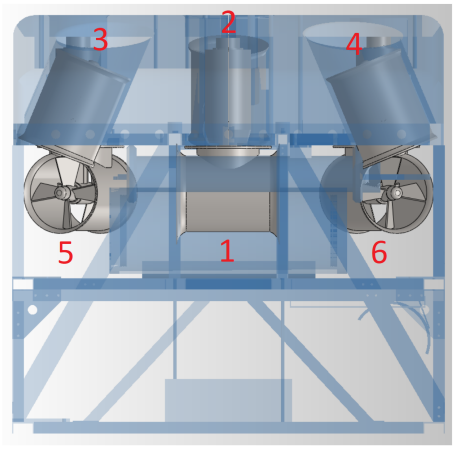
where $\boldsymbol{\eta}$ is the vector composed of the position $\boldsymbol{x} = (x, y, z)$ (expressed in the Gazebo reference frame) and the Euler angles (roll, pitch and yaw) $\boldsymbol{\Theta} = (\phi, \theta, \psi)$, $\boldsymbol{\nu}$ is the vector



(a) Thrusters layout of the RexROV 2 (top view).



(b) Thrusters layout of the RexROV 2 (side view).



(c) Thrusters layout of the RexROV 2 (front view).

Figure 6.5. Multiple point of views of the thrusters layout of the RexROV 2.

composed of the linear velocities $\boldsymbol{v} = (v_x, v_y, v_z)$ and the angular velocities $\boldsymbol{\omega} = (\omega_x, \omega_y, \omega_z)$, and $\boldsymbol{u}$ is the vector composed of the control and propulsion forces composed of the control inputs u_i sent to the six thrusters of the RexROV 2.

6.1.2.2 The control task

AUVs can be employed for various types of missions or control tasks: path planning, obstacle avoidance, waypoint tracking, station keeping, etc. In this work, we chose to make the RexROV 2 perform a waypoint tracking task. The task will be only operated in simulation, since we did not have the time to test the algorithms on real AUVs. This waypoint tracking problem aims to compare a deep-reinforcement-learning-based controller with a PID controller. Following the RL terminology, the task is divided in a finite number of episodes. In each episode, the AUV must reach a different 3D point called the waypoint or target waypoint. These episodes are composed of time steps, and the maximum number of time steps has to be tuned for each RL task.

At the beginning of each episode, we initialize the AUV at the 3D position $\mathbf{x} = [0, 0, -20]$ (in meters and relative to the frame attached to the Gazebo world center) and we assign it a random orientation $\boldsymbol{\Theta}$. More precisely, the Euler angles are initialized with $\phi = \theta = 0$ and ψ taken randomly in the range [0: 360].
At the beginning of each episode, a waypoint is randomly placed inside a bounded 3D box in the world of the simulator, centered on the position $[0, 0, -20]$. The size of the box can vary depending on the difficulty we want to assign to the control task. On the Z axis, the box will always take the range [-60:-1]. For the X and the Y axes, two cases are possible:

- **The simpler task:** the box will take the range [-20:20] on both X and Y axes.
- **The harder task:** the box will take the range [-50:50] on both X and Y axes.

The Figure 6.6 shows these two cases, represented in the horizontal plane (O,X,Y) (since the two boxes have the same range for the Z axis). The simpler task will be used in the section 6.3, while the harder task will be used in the sections 6.2 and 6.4.

During the execution of a training or testing episode, the AUV cannot exceed the vertical boundaries [-60:-1] of the Z axis. If it does so, the episode ends with a failure called a *collision*. If the AUV is able to reach a distance of less than 3 meters from waypoint without going outside of the vertical boundaries, the episode ends with a success. Finally, if the AUV spends too much time steps without reaching the waypoint, the episode ends with a failure (we call this case of failure a *timeout*): we empirically set the maximum number of time steps to 1000, which is a sufficient duration for the AUV to reach the target waypoints of both the simpler and the harder tasks.

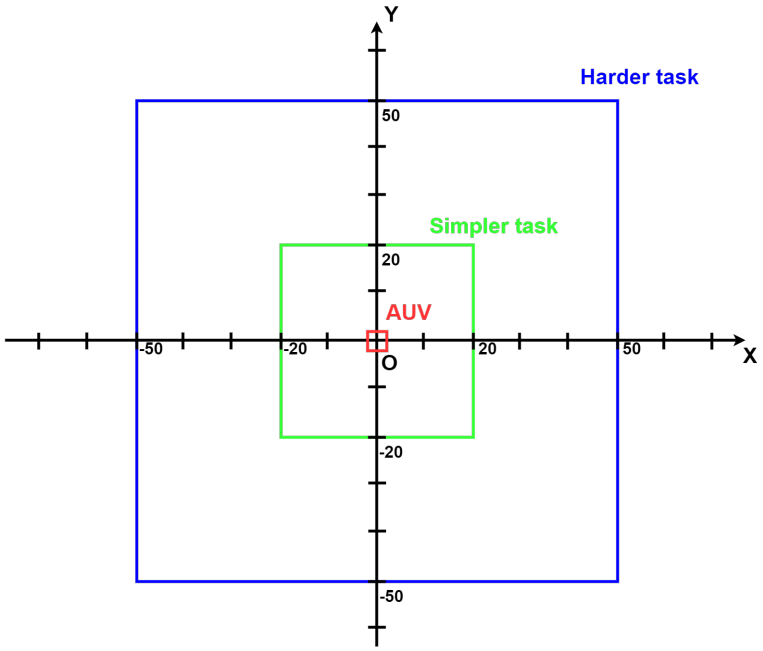


Figure 6.6. The bounded boxes of the simpler and the harder tasks represented in the horizontal plane (O,X,Y).

In order to make the simulation more realistic, we added noise to the sensors measurements of all the state variables of the RexROV 2. For each variables, a noise σ_i is added to its original value by randomly sampling the interval $[0.05 : 0.1]$ following a uniform distribution. A random noise σ_i is also added to each components of the action vector $\boldsymbol{u}$ by uniformly sampling the interval $[0.01 : 0.05]$.
We also added external disturbances to our underwater simulated environment thanks to the use of fluctuating ocean currents. Each 100 timesteps, the ocean current velocity $c_v \in [0 : 1]$ (in $m.s^{-1}$) and the ocean current angles $(c_{ha}; c_{va}) \in [-0.5 : 0.5]$ (for horizontal and vertical angles respectively in *radians*) are randomly modified by uniformly sampling a new value in their respective ranges. The Figure 6.7 shows an example of ocean current vector in the Gazebo reference frame. The fact that the angles and the velocity of the ocean currents are constantly changing during the execution of the episodes adds a real challenge to the control task.

The UUV Simulator offers several underwater environment files, called *worlds*, which can be either fictional (Empty underwater world, AUV underwater world, Ocean waves world, Lake) or based on real-world locations (Herkules ship wreck, the coast of Mangalia in Romania, Munkholmen in Norway, Subsea BOP panel). We chose to use the *Ocean waves world* which is a generic underwater environment providing a realistic seabed with a great variability of

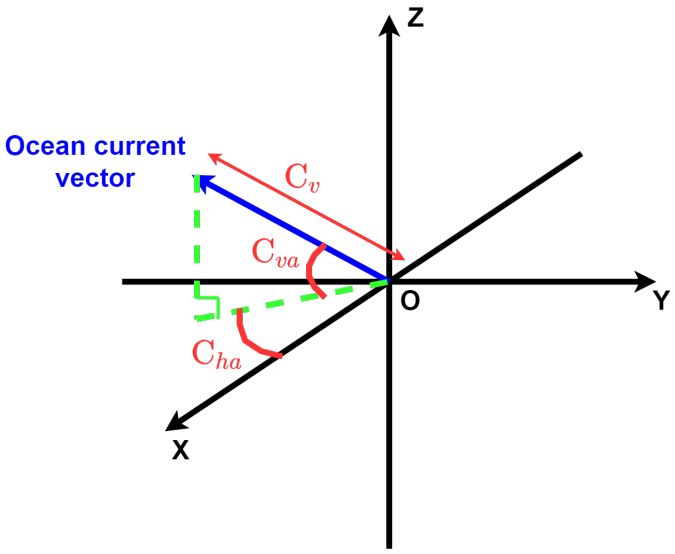


Figure 6.7. An example of ocean current vector in the Gazebo reference frame.

reliefs.

This work has been focusing on the low-level control of the AUV as well as the integration of guidance functions: the deep RL-based controller has to perform what is called an end-to-end control. This means that the navigation component of the GNC system (see section 2.1.2) is kept relatively simple for this control task and is not the subject to specific studies. The navigation component is provided by Gazebo, which simply sent the true values of the AUV variables with added noises.

The Figure 6.5 shows the thrusters layout of the RexROV 2. We can see that some of its thrusters can affect multiple degrees of freeedom (DOF) at the same time, because of their orientation. The DOFs of this AUV are thus strongly correlated, making the control task even harder to perform. Other stable cube-shaped AUVs would be able to move horizontally or vertically using independant thrusters, but the RexROV 2 required specific thrusters combinations for each movement. Understanding the dynamics of this AUV represents a real challenge for any learning algorithm.

6.1.3 Implementation of the control algorithms and definition of the metrics

In this section, we are going to describe the algorithms we are going to use for the control of the RexROV 2 as well as the metrics allowing to judge their respective performance. The task described in the section 6.1.2.2 will be performed by both the Soft Actor-Critic (SAC)

algorithm (see section 3.2.3) and a PID controller (see section 2.2) using the UUV Simulator. For each trial on this task, the procedure will be the following:

1. **Training phase:** the SAC algorithm is trained on successive training episodes. Each episode will replicate the task of the section 6.1.2.2;, with a randomly placed target waypoint and random time-varying ocean currents. The SAC-based controller will be trained until reaching the pre-defined maximum amount of training episodes. The learning process can also be manually cut in order to save computational time. The training phase can be stopped if we observe that the SAC algorithm fail to learn the task by not reaching a lot of waypoints. At the contrary, it can also be stopped if we judge that the SAC-based controller has already achieved satisfactory results and cannot learn anything more from the training episodes.

2. **Choosing the trained models:** During the training phase, the parameters of the neural networks of the SAC are constantly evolving. The values of these parameters are regularly saved and stored in specific files during the learning process. We chose to save these parameters every 100 training episodes until the episode number 1500, and then every 250 episodes until the episode number 5000 (we fixed the maximum amount of training episodes to 5000). All these saved parameters form a set of *trained models*. For example, the parameters saved at the episode 1000 will be denoted the *model 1000*. Based on the amount of successes obtained during the training phase, we will choose one or more trained model in order to compare them with the PID controller.

3. **Testing phase:** the chosen trained models are independently compared with a PID controller. Each trained model is run on a distinct testing phase of 1000 episodes, composed of 500 test episodes running the SAC-based controller and 500 test episodes running the PID controller. These test episodes reproduce the same task as during the training phase, but now the parameters of the neural networks cannot vary anymore (since the learning process is over). The SAC and the PID controllers have to reach the same set of random target waypoints and are subject to the same time-varying ocean currents, in order to preserve the consistency of this comparison.

4. **Analyzing the results:** Once the testing phase is finished, the results are analyzed thanks to different performance metrics. These metrics are computed a posteriori using Matlab.

6.1.3.1 Implementation of the PID algorithm

The UUV Simulator proposes several ready-to-use implementations of classical controllers: PID controllers, PD controllers with compensation of restoring forces for dynamic positioning, model-free sliding mode controllers based on [93] and [283], model-based sliding mode controllers, model-based feedback linearization controllers, singularity-free tracking controllers

based on [84]. We chose to use the PID controller for the comparison with the SAC algorithm since it is the most used algorithm in the control theory literature.

The PID controller is defined as a multi-input-multi-output (MIMO) system and is able to regulate the six degrees of freedom of the RexROV 2 in order to reach the waypoint: the surge, the sway and the heave, as well as the roll, the pitch and the yaw (as defined in the section 4.1). It computes an input vector $\mathbf{u}$ based on the tracking error $\mathbf{e(t)} = \mathbf{r(t)} - \hat{\mathbf{y}}$, with $\mathbf{r}$ being the reference (corresponding here to the waypoint) and $\hat{\mathbf{y}}$ being a measurement of the output variables of the AUV. This vector is defined as follows:

$$\boldsymbol{u}(t) \;=\; \boldsymbol{K_p}\; \boldsymbol{e}(t) \;+\; \boldsymbol{K_i}\; \int_0^t \boldsymbol{e}(\tau) d\tau \;+\; \boldsymbol{K_d}\; \frac{d\boldsymbol{e}(t)}{dt} \tag{6.2}$$

with

$$\boldsymbol{e}(t) \;=\; [x_e, y_e, z_e, \phi_e, \theta_e, \psi_e]^T \tag{6.3}$$

where $\boldsymbol{K_p}$, $\boldsymbol{K_i}$ and $\boldsymbol{K_d}$ are real-value matrices called the *gains*, (x_e, y_e, z_e) are the errors between the waypoint position and the AUV position (expressed in meter, in Cartesian coordinates and in the absolute reference frame of gazebo) and $(\phi_e, \theta_e, \psi_e)$ are the Euler angles errors corresponding to the amount of angles the AUV is lacking in order to point towards the waypoint (expressed in radians, in the local reference frame of the AUV).

The PID controller provided by the RexROV 2 package is already tuned and its gains were found using SMAC (Sequential Model-based optimization for general Algorithm Configuration) [145] [15]. $\boldsymbol{K_p}$, $\boldsymbol{K_i}$ and $\boldsymbol{K_d}$ are diagonal matrices defined with the following diagonal coefficients (here they have been rounded for more clarity):

$$\begin{cases} \boldsymbol{diag\_K_p} &= [11994, 11994, 119934, 19460, 19460, 19460]^T \\ \boldsymbol{diag\_K_i} &= [321, 321, 321, 2097, 2097, 2097]^T \\ \boldsymbol{diag\_K_d} &= [9077, 9077, 9077, 18881, 18881, 18881]^T \end{cases} \tag{6.4}$$

These inputs $\mathbf{u}$ are not sent directly to the AUV thrusters. More specifically, the vector $\boldsymbol{u}$ can be written as $\boldsymbol{u} = (f_x, f_y, f_z, \tau_r, \tau_p, \tau_y)^T$, where (f_x, f_y, f_z) are forces and (τ_r, τ_p, τ_y) are torques. The forces and torques provided by the PID controller need to be applied to the AUV in its local reference frame. The UUV Simulator uses an intermediary component called the *thruster manager* which allows to transforms these input or forces/torques signals given by the PID controller into thrusters commands. In the case of the RexROV 2, there are six thrusters operating the AUV. As said earlier, each thruster can affect a combination of several degrees of freedom. The thruster manager defines a *Thruster Allocation Matrix (TAM)* in order to transform the PID input vector $\mathbf{u}$ into a thruster commands vector $\mathbf{c}$ as follows:

[15] https://github.com/automl/SMAC3

$$\mathbf{c(t)} \;=\; \mathbf{TAM} \;.\; \mathbf{u(t)} \tag{6.5}$$

The RexROV 2 package gives the following 6x6 **TAM** (here the coefficients have been rounded to three digits for more clarity):

$$\begin{pmatrix} 0.0 & 0.259 & 0.0 & 0.0 & 0.906 & 0.906 \\ 0.999 & 0.0 & 0.383 & 0.383 & 0.423 & -0.423 \\ 0.0 & 0.966 & 0.924 & -0.924 & 0.0 & 0.0 \\ -0.237 & 0.0 & -0.696 & -0.696 & -0.102 & 0.102 \\ 0.0 & 0.946 & -0.799 & 0.799 & 0.218 & 0.218 \\ 0.488 & 0.0 & 0.331 & 0.331 & -0.764 & 0.764 \end{pmatrix} \tag{6.6}$$

Each line of the **TAM** corresponds to a different thruster of the RexROV 2. After this computation, each component of the command vevtor $\boldsymbol{c}$ is sent to its respective thruster, following the same indexing as shown on the Figure 6.5. The thruster manager also allows to change specific parameters such as the maximum thrust which can be applied to the AUV, the update rate or the type of the thrusters.

The guidance algorithm used with the PID controller will simply consist in straight lines going from the initial position of the RexROV 2 to the target waypoints. No navigation algorithm will be used: the variables will be sent directly to the PID by the UUV Simulator, with added noises (described in the section 6.1.2.2).

6.1.3.2 Implementation of the Soft Actor-Critic algorithm for AUV control

The deep reinforcement learning (deep RL) algorithm we chose to evaluate for the waypoint tracking task of the RexROV 2 is the Soft Actor-Critic (SAC), which was previously detailed in the section 3.2.3. It is a model-free approach belongings to the family of Policy Gradient (PG) technics. We chose the SAC because it is one of the latest advances in deep RL and because it has never been used for the control of an AUV. Moreover like a lot of deep RL technics, it is very appropriate for continuous control tasks thanks to the use of neural networks, which allows to work in continuous state and action spaces (as it is the case in robotics).

In this section, we will not detail the whole implementation of the algorithm, and we will only explain the specific tuning of the SAC for our use case. Moreover some of the parameters presented here may vary throughout the next sections, and the version of the SAC in this section can be considered as the base version of this algorithm.

In reinforcement learning (see more details in 3.2), an agent (here the SAC algorithm) evolve in an unknown environment (the underwater environment) in order to fulfill a task (reaching

waypoints). At the time step t, the agent applies an action a_t to the environment, while the environment responds to that with a feedback composed of a state s_t and a reward signal r_t. The state is a partial observation of the environment, while the reward judge how good the agent is acting with respect to the given task. This sequence is repeated until the episode ends. More specifically, the goal of the agent is to maximize the return (the total sum of discounted rewards). In the case of the SAC algorithm, the agent must also maximize an entropy term in order to encourage the exploration of the environment. The global objective function which to needs to be maximize by the agent is:

$$J(\boldsymbol{\theta}) = \sum_{t=0}^{T} \gamma^t \, \mathbb{E}_{(s_t,a_t)\sim\rho_\pi}[r(s_t,a_t) + \alpha\mathcal{H}(\pi_{\boldsymbol{\theta}}(.|s_t))] \tag{6.7}$$

where α is temperature parameter controlling the maximization of the entropy, γ is the discount factor allowing to less rely on uncertain future expected rewards, and ρ_π is the marginal of the state distribution induced by the policy $\pi_{\boldsymbol{\theta}}(a|s)$, as defined in the DPG section. γ is set to the value 0.99, while the value of α is set according to the scale of the reward function and will be described at the end of this section.

Policy Gradient-based models are particularly sensitive to the shape of the reward function. The smallest variations can lead to the convergence or not of the model, and the magnitude and the evolution of each term composing the reward function must be chosen carefully. This function is often crafted by trial and error, since no global rules exist in the reinforcement learning theory, and the expression of the function can drastically change from one task to another. The simulated environment used during the training must be realistic enough, while not being too complex for the agent. Indeed, if the task requested is too complex, the agent might not be able to converge: for example if the state returned by the environment is not descriptive enough, or if the reward function is too constraining. A trade-off between the reward function and the complexity of the environment must therefore be found.
Since our study involve MIMO dynamical systems, the state s_t and the a_t will be vectorial from now.

Unlike the PID controller, the SAC does not use a thruster manager: it outputs an action vector $\mathbf{A}_t = [u_1, u_2, u_3, u_4, u_5, u_6]^T$ which is composed of the inputs u_i sent directly to the six thrusters of the RexROV 2. Each input belongs to the range $[-240.0 : 240.0]$.
At the time step t of a given episode, the environment sent the following state vector $\mathbf{S}_t$ to the SAC algorithm, composed of 23 variables:

$$\mathbf{S}_t = [\ \boldsymbol{x},\ \boldsymbol{\Theta},\ \boldsymbol{v},\ \boldsymbol{\omega},\ \theta_e,\ \psi_e,\ \boldsymbol{x}_e,\ \boldsymbol{u}_{t-1}\ ]^T \tag{6.8}$$

where $\boldsymbol{x} = (x, y, z)$ is the position vector of the RexROV 2, $\boldsymbol{\Theta} = (\phi, \theta, \psi)$ is its orientation vector (the Euler angles), $\boldsymbol{v}$ is the linear velocity vector (the derivative of $\boldsymbol{x}$), $\boldsymbol{\omega}$ is the angular velocity vector (the derivative of $\boldsymbol{\Theta}$), θ_e and ψ_e are respectively the tracking errors of the pitch

and the yaw angles (the amount of angles that is lacking in order to make the AUV point towards the waypoint), $\boldsymbol{x}_e$ is the error between the position vector $\boldsymbol{x}$ and the position vector of the waypoint, and $\boldsymbol{u}_{t-1} = (u_1, u_2, u_3, u_4, u_5, u_6)$ is the action vector of the inputs sent at the previous time step.

In order to perform the waypoint tracking mission, we designed the reward function r_t defined in (6.9). It was inspired partly by [51], where the reward function takes into account low-level variables such as linear and angular velocities and their respective references. We adapted this work with other control variables, taking directly into account the position of the AUV and its reference, and by setting a lot of the values empirically:

$$r_t = \begin{cases} r_{waypoint} & \textbf{if } \ d_t < \epsilon \ \ \text{(success)} \\ r_{collision} & \textbf{if } \ z \notin [z_{min} : z_{max}] \ \ \text{(collision)} \\ r_{toward} & \textbf{if } \ d_t < d_{t-1} \\ r_{backward} & \textbf{if } \ d_t \geq d_{t-1} \end{cases} \tag{6.9}$$

where r_t is the reward received by the agent at time t, d_t is the current relative distance between the AUV position and the position of the waypoint to reach, z_{min} and z_{max} are the authorized limits for the vertical movement z of the AUV, and ϵ is the threshold allowing to detect when the AUV has reached the waypoint (the AUV does not need to reach exactly the waypoint, and is asked to enter inside a sphere of radius equal to ϵ meters). As described in the section 6.1.2.2, z_{min} is set to -60 meters and z_{max} to -1 meter, whereas ϵ is set to 3 meters. Each term appearing in (6.9) represents a specific feature of the global desired behavior of the AUV:

- $r_{waypoint}$ is a constant positive reward given to the agent when the AUV reaches the waypoint, which leads to a terminal state, ending the current episode with a success. It is set to 500.

- $r_{collision}$ is a constant negative reward given to the agent when the vertical movement of the AUV exceeds the limits defined by $[z_{min} : z_{max}]$ (here [-60: -1]), which leads to a terminal state, ending the current episode with a failure. It is set to -550.

- r_{toward} is a variable reward given when the distance d_t is decreasing, which means that the AUV moves toward the waypoint. It is defined as follows:

$$r_{toward} \ = \ \lambda_1(d_t - d_{t-1}) \ - \ \lambda_2 \ \|\Omega\| \tag{6.10}$$

  where λ_1 and λ_2 are positive weighting terms set respectively to 200 and 10. The term weighted by λ_1 rewards large movements toward the waypoint, while the term weighted by λ_2 penalizes strong angular speeds, and promotes indirectly a softer use of the actuators.

- $r_{backward}$ is a constant negative reward given when the distance d_t is increasing or staying equal to its previous value, which means that the AUV moves backward the waypoint or stays still. It is set to -10.

We recall that the SAC algorithm is based on an Actor-Critic architecture and is composed of four neural networks:

- The soft state-value network $V_{\boldsymbol{\psi}}$ (or sometimes simply called soft value network), modeled by the parameters $\boldsymbol{\psi}$. The input of the network as a size of 23 and is composed of the state $\mathbf{S}_t$ defined in 6.8, while its output is the state-value function $V(\mathbf{S}_t)$. This function allows to evaluate how good the state $\mathbf{S}_t$ is. This neural network is composed of two hidden layers of 256 neurons and one ouput layer composed of 1 neuron (corresponding to the estimation of the scalar $V(\mathbf{S}_t)$). The activation function used in the two hidden layers is the leaky ReLU (Rectified linear unit), the classical deep learning activation function (mentionned in 3.1.1) defined as $f(x) = \max(0.01x, x)$. The output layer has no activation function and remains a linear layer in order to perform the regression of the state-value function.

- The target network of the soft state-value network, modeled by the parameters $\boldsymbol{\psi}'$. It is used in the computation of the loss function of the soft state-value network.

- The soft Q-value network $Q_{\boldsymbol{w}}$, modeled by the parameters $\boldsymbol{w}$. The input of the network as a size of 29 and is composed of the state $\mathbf{S}_t$ and the action $\mathbf{A}_t$, while its output is the Q-value function $Q(\mathbf{S}_t, \mathbf{A}_t)$. This function allows to evaluate how good the action $\mathbf{A}_t$ is, with rescpect to the state $\mathbf{S}_t$. This neural network is composed of two hidden layers of 256 neurons and one ouput layer composed of 1 neuron (corresponding to the estimation of the scalar $Q(\mathbf{S}_t, \mathbf{A}_t)$). The activation function used in the two hidden layers is the leaky ReLU. The output layer has no activation function and remains a linear layer in order to perform the regression of the state-value function.

- The policy network $\pi_{\boldsymbol{\theta}}$, modeled by the parameters $\boldsymbol{\theta}$. The input of the network as a size of 23 and is composed of the state $\mathbf{S}_t$ defined in 6.8, while its output is composed of the mean vector $\boldsymbol{\mu}_{\boldsymbol{\theta}}$ and the variance variance $\boldsymbol{\sigma}_{\boldsymbol{\theta}}$. These vectors are used to compute the action vector $\mathbf{A}_t$ as follows:

$$\mathbf{A}_t = \tanh(\boldsymbol{n}) \quad \text{where } \boldsymbol{n} \sim \mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{\theta}}, \boldsymbol{\sigma}_{\boldsymbol{\theta}}) \tag{6.11}$$

  This neural network is composed of two hidden layers of 256 neurons and one ouput layer composed of 12 neurons: 6 neurons for the components of the vector $\boldsymbol{\mu}_{\boldsymbol{\theta}}$ and 6 neurons for the components of the vector $\boldsymbol{\sigma}_{\boldsymbol{\theta}}$. The activation function used in the two hidden layers is the leaky ReLU. The output layer has no activation function and remains a linear layer in order to perform the regression of the two mean and variance vectors.

All the components of the weight matrices and the bias vectors of these neural networks are randomly initialized using a uniform distribution over the range $[-3.10^{-3} : 3.10^{-3}]$, except for the target network: its parameters are initialized using a copy of the initial parameters of the soft state-value network.
The parameters of the target network are updated using the rule: $\boldsymbol{\psi}' \leftarrow \tau\boldsymbol{\psi} + (1-\tau)\boldsymbol{\psi}'$, where τ is set to 5.10^{-3}. The three other neural networks are all updated thanks to the ADAM algorithm (detailed in the section B.3) and the loss functions defined in the section 3.2.3.3, using the same learning rate set to 3.10^{-4}. The updates are computed using batch of training samples taken from a replay buffer D able to contain up to 5.10^6 transitions. The size of the batches is set to 256. The temperature parameter α is set to the inverse of the reward range. In this task we find empirically that the reward function lays around the interval [-20: 20], which means a range of 40: we set the temperature to $\alpha = \frac{1}{40} = 0.025$. A lot of the previous hyperparameters has been set to the values recommended by [116], the original paper of the SAC algorithm. The authors kept using some of these values during a lot of different use cases.

The SAC algorithm has to perform an end-to-end control of the AUV. This means that it has to carry out both the low-level and high-level control (called guidance) of the AUV. It has to simulataneously send the good inputs to the thrusters of the AUV (low-level) and choose the right trajectories to follow (high-level).
As we are focusing on the low-level control and the guidance of the RexROV 2 and not on its navigation, the tracking errors of the variables are directly given inside the state vector $\mathbf{S}_t$. Since the SAC algorithm does not use a thruster manager and sends directly its actions as the inputs of the thrusters, the task is harder for this deep RL agent. Indeed, it has to figure out the dynamics of the RexROV 2 and how each thruster is affecting the different degrees of freedom of the AUV.

6.1.3.3 Definition of the performance metrics

When the testing phase is completed, the records of the test episodes of both controller are analysed using the software *Matlab*. We defined metrics in order to measure different aspects of the performance of these controllers such as the speed of convergence of the learning algorithms, the efficiency in the tracking of the waypoints, or even the energy consumption of the AUV.

The only metric used during the training phase of the SAC algorithm is called the *Number of success for each 100 episodes*. It counts the number of successful episodes obtained each one hundred training episodes: it counts the episodes in which the AUV reached the waypoint. It allows to measure how fast the SAC algorithm is able to reach an acceptable behaviour during its learning of the neural networks parameters. This metric will be presented with the following format: [5, 10, ... , 95, 100, 99]. We said earlier that the training phases are

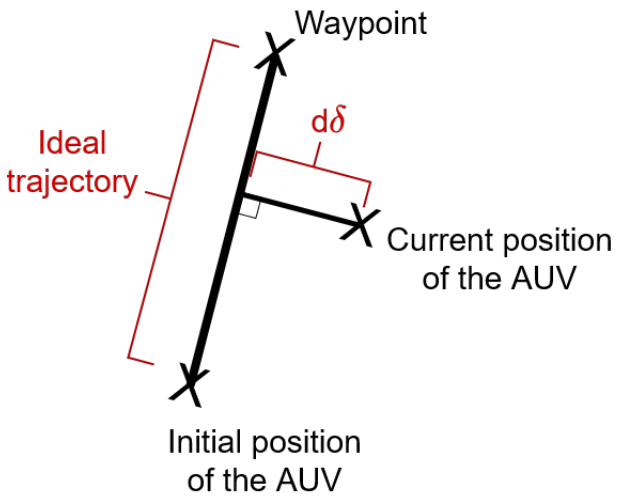


Figure 6.8. Definition of the ideal trajectory and the distance error $d\delta$.

composed of a maximum number of 5000 episodes, so the array of the *Number of success for each 100 episodes* can be composed of 50 numbers or less.

During the testing phase, more metrics were defined in order to compare the PID and SAC controllers according to different criteria. Here are the list of these metrics, all computed on data form the test episodes of both controllers (500 episodes for each of them):

- **Success rate:** the success rate is the percentage of successful episodes a controller got throughout its 500 test episodes. As defined in the section 6.1.2.2, a *success* happens when the AUV reaches the waypoint without going outside of vertical boundaries and without exceeding the maximum number of time steps.
- **Collision rate:** the collision rate is the percentage of unsuccessful episodes a controller got throughout its 500 test episodes because of collisions. As defined in the section 6.1.2.2, a *collision* happens when the AUV goes beyond the vertical boundaries [-60: -1] on the Z axis of the environment and the episode ends with a failure.
- **Timeout failure rate:** the timeout failure rate is the percentage of unsuccessful episodes a controller got throughout its 500 test episodes because of timeout failures. As defined in the section 6.1.2.2, a *timeout* happens when the AUV spends more than 1000 time steps without reaching the target waypoint and the episode ends with a failure.
- **Mean of** $d\delta$: We define the notion of *ideal trajectory* as follows: the ideal trajectory is the perfect path allowing the AUV to go directly to the waypoint. In practice, it corresponds to the straight line linking the initial position of the AUV and the target waypoint. We

also define the *distance error* $d\delta$ as the measure of the deviation of the AUV from the ideal trajectory. This deviation is expressed in meters and is measured using a 3D line perpendicular to the ideal trajectory and passing through the current position of the AUV (see Figure 6.8). The *Mean of* $d\delta$ is the mean value of the distance error $d\delta$, computed on all the time steps of all the test episodes.

- **SD of** $d\delta$: by keeping the previous definitions of the ideal trajectory and the distance error $d\delta$, *SD of* $d\delta$ is the standard deviation of the distance error $d\delta$, computed on all the time steps of all the test episodes. The mean and the SD of $d\delta$ allow to assess the tracking abilities of the controller and are both expressed in meters.

- **Mean of** $\|\mathbf{u}\|$**:** As defined earlier the vector $\mathbf{u}$ is composed of the six commands u_i received by the thrusters of the RexROV 2. Its Euclidean norm is defined as $\|\mathbf{u}\| = \sqrt{({u_1}^2 + {u_2}^2 + {u_3}^2 + {u_4}^2 + {u_5}^2 + {u_6}^2)}$. The *Mean of* $\|\mathbf{u}\|$ is the mean value of the norm $\|\mathbf{u}\|$, computed on all the time steps of all the test episodes. The mean of $\|\mathbf{u}\|$ gives an idea of the global thrusters usage made by the controller. The bigger the commands u_i are, the more the thrusters will be used. This metric gives indirectly an information about the durability of an AUV controlled by a given controller: they smaller the demand on the thrusters is, the more the AUV actuators will last.

- **Mean number of steps:** The *Mean number of steps* is a metric measuring the mean number of time steps taken by a test episode of a given controller. It simply gives the mean duration of the episodes performed during the testing process, independently of the result of these episodes (success or failure). The mean number of steps is correlated to the mean and the SD of $d\delta$, since a great number of time steps taken by an episode can be due to the fact that the AUV is deviating too much from the ideal trajectory, and that it has struggled to reach the target waypoint.

- **Mean of** $\sum \|\mathbf{u}\|$**:** For each episode, the sum of the norm $\|\mathbf{u}\|$ is computed, noted $\sum \|\mathbf{u}\|$. It gives a precise idea of the total amplitude of all the commands asked to the thrusters during each episode. The *Mean of* $\sum \|\mathbf{u}\|$ is simply the mean value of the quantity $\sum \|\mathbf{u}\|$, computed on all the test episodes. This metrics gives the mean thrusters usage per episode, and gives indirectly an information about the energy consumption of the AUV: the greater the quantity $\sum \|\mathbf{u}\|$ is for one episode, the bigger the energy consumption of the AUV was during that episode. Indeed, the greater the command u_i is, the greater the electric current sent to the thruster i will be. The values of the mean of $\sum \|\mathbf{u}\|$ shown in the results of the next sections will always have to be multiplied by 10^5 in order to have the true numbers. We multiplied the values of the mean of $\sum \|\mathbf{u}\|$ by 10^{-5} in order to not have too large numbers in our results tables.

The result of the test episodes can be either a success or a failure and can biased the metrics based on the distance error $d\delta$, the time steps and the norm $\|\mathbf{u}\|$, depending on what we want

to analysed. All the cited metrics have also been computed using only successful episodes, and these additional values are denoted with term *(Success)* in front of their name. For example, these *(Success)* versions of the metrics can be employed in order to know the mean time needed by the controllers to reach the waypoints, or the amplitude of the deviations from the ideal trajectory caused only by the ocean currents (and not caused by the aberrant behaviour of a machine learning-based controller that has not learnt well the task).
Finally, all the metrics will not be systematically commented. We will display tables giving the maximum amount of information to the reader, but we will only discuss the most relevant values for each trial. Moreover, some metrics are also implicitly included in one another, and can be considered as intermediary values. For example, the mean of $\sum \|\mathbf{u}\|$ can be viewed as an approximation of the product of the mean of $\|\mathbf{u}\|$ and the mean number of steps: it is the expression of an energy value and is the product of a time value and an intensity value.

6.2 Initial trial on the waypoint tracking task

In this section, we describe the result of our first trials of applying the SAC to the control of the RexROV 2 on a waypoint tracking task. More precisely, the goal of this first task was reaching waypoints randomly placed inside a large box with the ranges [-50: 50] on the X and Y axes and [-60: -1] on the Z axis (see the section 6.1.2.2). These results were published in [309].

Firstly, we trained the SAC algorithm on training episodes (with a maximum of 5000 episodes). We had the following number of success during the training process:

Number of success for each 100 episodes: [91, 95, 99, 100, 99, 100, 100, 100, 100, 100, 99, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 61, 7, 5, 3, 53, 34, 0, 0, 22, 81, 91, 100, 100, 99, 100, 100, 100, 83, 98, 91, 13]

We stopped the training after 4600 episodes and we can see that the SAC algorithm rapidly converged, since it managed to get 91 successes during its first 100 episodes. This is a really rapid convergence towards a good behaviour and confirms that the SAC algorithm can be really sample efficient. We can note that after 2500 episodes, the number of success begin to drop drastically before going up again. This could come from the maximization of the entropy term by the SAC, which allows to favor exploration of unknown parts of the environment. As said in the previous section, the parameters of the SAC are regularly saved inside specific files, forming a set of trained models.
Then, the SAC has been compared with the PID controller during 1000 episodes, giving 500 test episodes for each algorithm. We chose to test the model obtained after 1200 training episodes (1200 episodes took around 4 hours to be performed.) and this results are shown on the Table 6.1.

Metrics	PID	SAC
Success rate (%)	96.0	86.4
Collision rate (%)	0.8	1.2
Timeout failure rate (%)	3.2	12.4
Mean of $d\delta$ (m)	3.81	8.69
SD of $d\delta$ (m)	3.53	5.47
Mean of $\|\mathbf{u}\|$	541.42	481.16
Mean number of steps	299	365
Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.618	1.756
(Success) Mean of $d\delta$ (m)	2.84	6.40
(Success) SD of $d\delta$ (m)	2.62	3.46
(Success) Mean of $\|\mathbf{u}\|$	540.47	472.84
(Success) Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.477	1.256

Table 6.1. Results for the initial waypoints tracking task, with waypoints placed inside a large box and with a model trained on 1200 episodes.

For each metric, we highlighted the controllers with the best results using a case filled with the green color.[16] We can see on the Table 6.1 that the PID controller had a greater number of success (96.0%) than the SAC algorithm (86.4%), and with less collisions and timeout failures. We recall that an episode ends with a success when the AUV goes inside a sphere of radius 3 meters centered on the waypoint. Collisions happen when the RexROV 2 exceeds the vertical limits -60 and -1 on the z axis, while timeouts happen when the RexROV 2 stays 1000 time steps inside the box without reaching the waypoint (the episode ends with a failure in both cases).
If we consider all the test episodes, the PID controller is consuming less energy (regarding the mean of $\sum \|\mathbf{u}\|$), even if the SAC algorithm is generating smaller mean inputs $\|\mathbf{u}\|$. This due to the fact that the SAC is deviating too much from the ideal trajectory leading to the waypoint, measured by the mean and the standard deviation of the error $d\delta$. It is also spending too much time steps during the episodes. However if we only take into account the successful episodes, the SAC algorithm is consuming less energy than the PID. This is really encouraging and the next steps will be to improve the success rate of the SAC algorithm.

An example of sucessful episode for both controllers has been chosen, and the trajectories followed by the RexROV 2 are shown in 3 dimensions on Figure 6.9 and in 2 dimensions

[16] The metrics called *(Success) Mean number of steps* was not computed for this first trial, since some variables required for its computation have not been saved in our logs at this time.

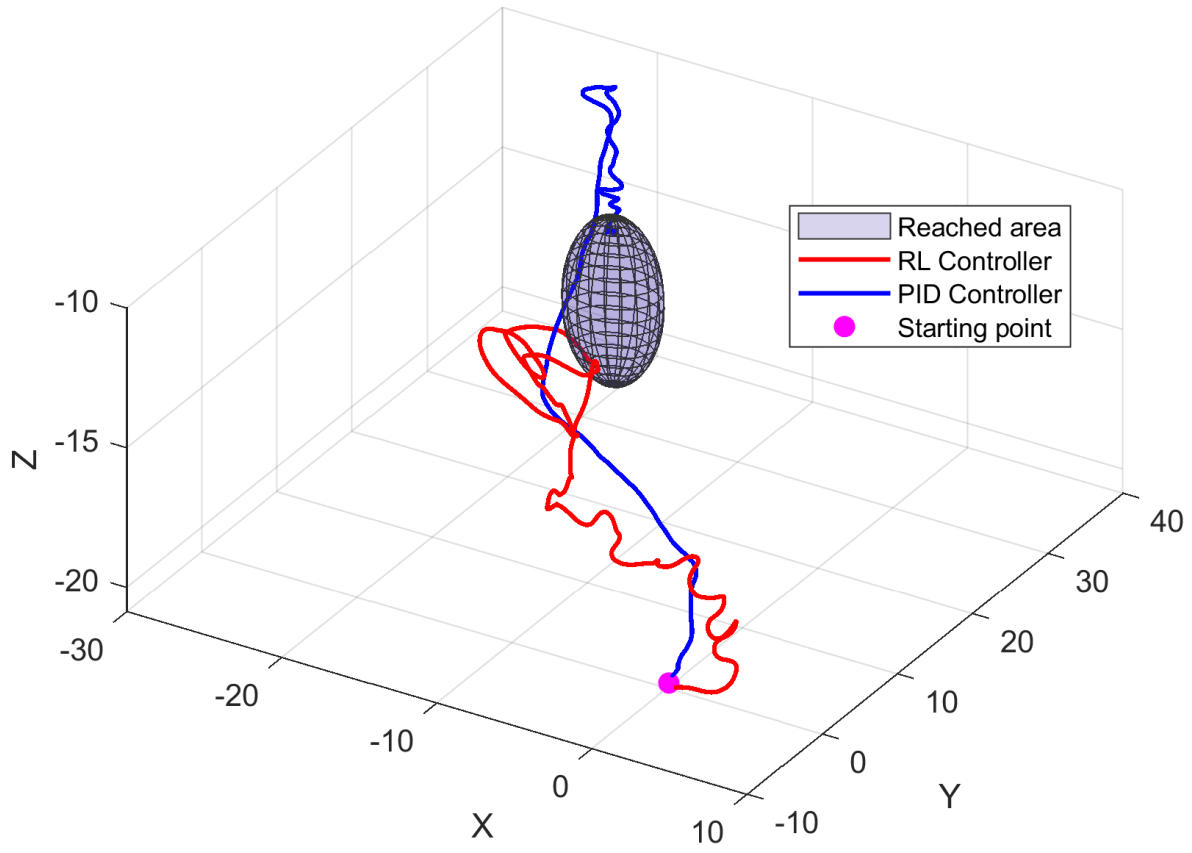
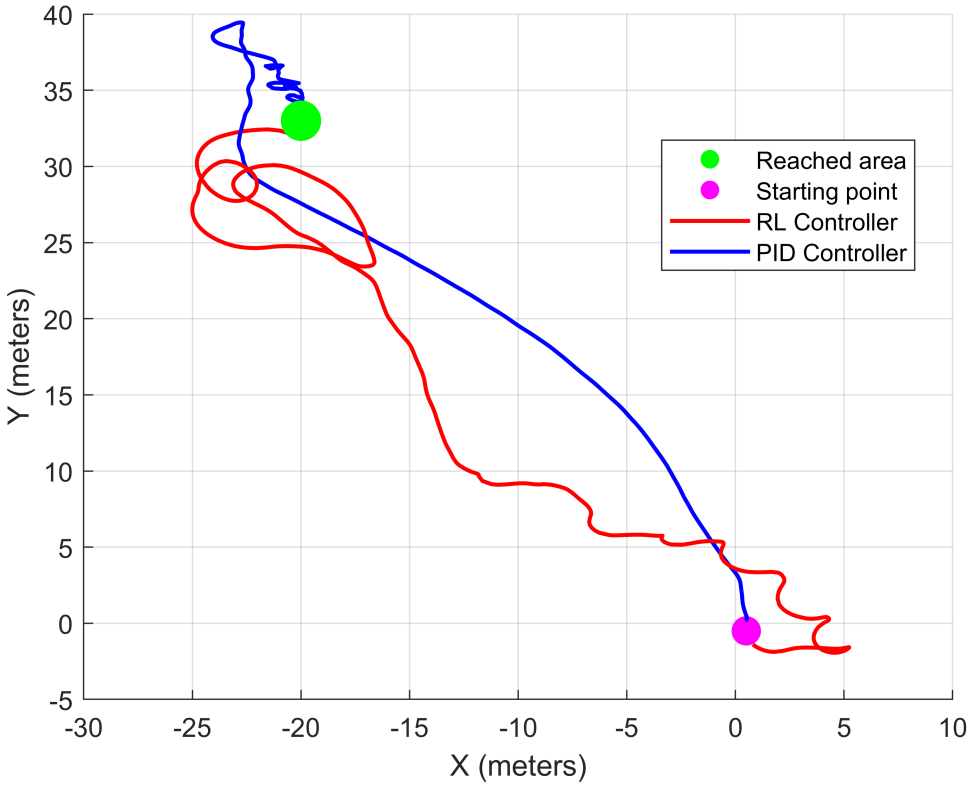


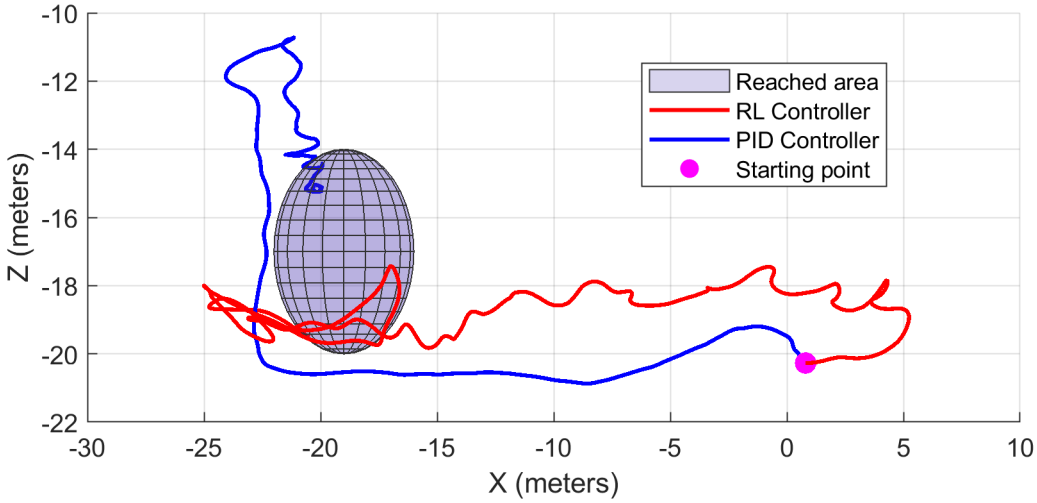
Figure 6.9. Example of 3D trajectories followed by the AUV for both of its controllers.

on Figure 6.10. On Figure 6.10a, the target waypoint is shown has a point, and we have represented a sphere with a 3 meters radius and centered on the waypoint on Figures 6.9 and 6.10b. When the AUV enters inside this sphere, the episode ends with a success. For the same episode, the evolution of the norm of the input vector $\mathbf{u}$ sent to the thrusters is shown on Figure 6.11, as well as the evolution of the distance error $d\delta$ on Figure 6.12.

We can see on the figures 6.9 and 6.10 that both controllers struggled to reach the waypoint directly, because their respective trajectory deviated from the ideal trajectory (the straight line connecting the waypoint to the initial position of the AUV) before finally reaching the sphere with a radius of 3 meters. **This is due to the successive changes appearing in the directions and the magnitude of the ocean currents each 50 time steps.** Moreover the Figure 6.12 shows that until the time step 400, both controllers have a distance error $d\delta$



(a) Trajectories according to the $(\vec{X}, \vec{Y})$ plane.



(b) Trajectories according to the $(\vec{X}, \vec{Z})$ plane.

Figure 6.10. Example of a 2D trajectory followed by the AUV for both controllers.

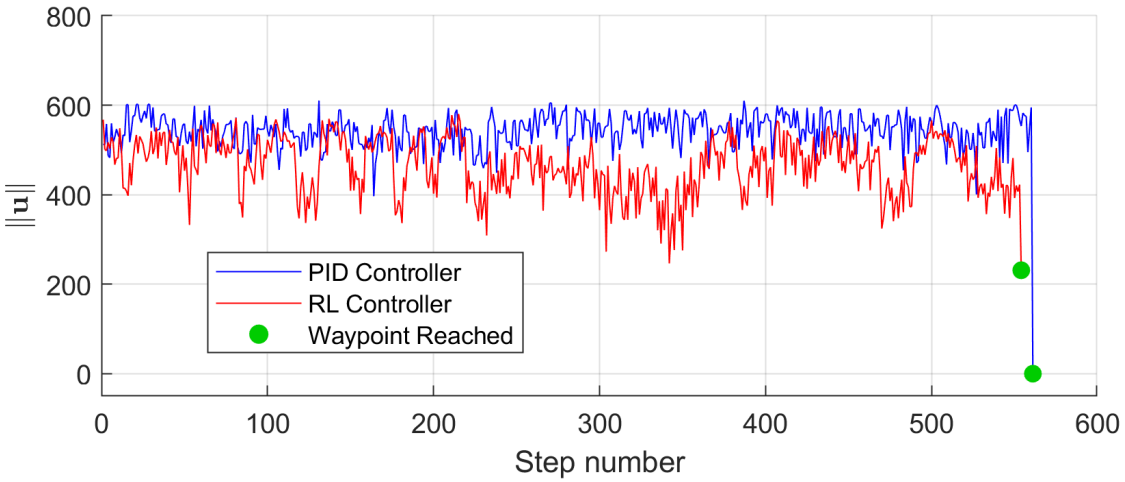


Figure 6.11. Euclidean norm of the input vector $\mathbf{u}$ over time steps.

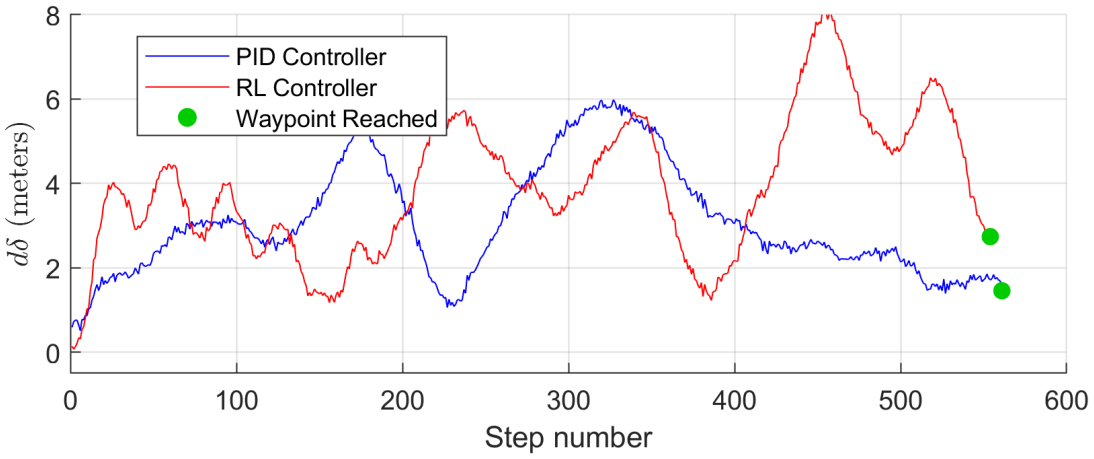


Figure 6.12. Distance error $d\delta$ of the AUV from its ideal trajectory over time steps.

relatively similar. After the time step 400, the SAC-based controller is deviating more from the ideal trajectory than the PID controller. Finally we can see on the Figure 6.11 that the euclidian norm of the input vector **u** generated by the PID controller is most of the time greater than the one generated by the SAC, explaining why the SAC-based controller is saving more energy than the PID.

6.3 Study of the impact of the state vector on a simpler task

After the previous initial trial where the PID outperformed the SAC in terms of success rate, we chose to slightly simplify the task in order to eventually make the SAC equalize the success rate of the PID controller. The difference is that now the waypoints appear inside a box with the ranges [-20: 20] on the X and Y axes and [-60: -1] on the Z axis. In order to avoid making this control task too easy, we also made the collision failures of the episodes happen when the AUV goes outside of the range [-30: 30] for the X and the Y axes and [-60: -1] for the Z axes. In this section, we will describe several definitive changes made to the simulation, and we will show the results of a study of the impact of the state vector components on the performance of the SAC on this new simplified task.

6.3.1 Improvements of the simulation setup

We decided to made some changes in order to improve our simulation setup, which will be kept for all the upcoming results in the next sections.
We first made the simulations to be computed in real time. In the previous section, the UUV Simulator had no limitations in its speed of computation and was able to perform the simulation with a factor of 4: it was able to compute four seconds of simulation during one real time second. The UUV Simulator is now forced to follow the real time: it computes one second of simulation during one real time second. This choice was made in order to facilitate the transfer of our SAC-based controller inside a real robotic platform. Since the UUV Simulator is based on ROS, the SAC algorithm and the simulated environment are encapsulated inside separate components and their execution are not synchronised. With an environment simulated with a factor of 4, neither the rates of the sensors measures sent to the controller nor the duration separating two successive inputs sent to the thrusters will be the same as in real life. These differences bias the SAC algorithm and it would not be able to perform the control task if it were embedded on a real-world AUV: if the SAC is trained with a faster simulation, the effect of its actions will not affect the behaviour of the real AUV the way we would expect, since the inputs will not be executed by the thrusters for the same duration as it used to learn in the training process. These remarks were confirmed by the fact that the SAC-based controller did not have the same behaviour when we assigned different values to the speed factor of the simulator.

We also made a change in the reward function. The global structure of the reward is still the same as in the equation 6.9, and only the term r_{toward} has been modified:

$$r_{toward} = 40. \exp\left(-\frac{d_t}{20}\right) \tag{6.12}$$

where d_t is the distance between the position of the AUV and the waypoint at the time step t. The use of the exponential function was inspired by [51] and has the effect of giving very large rewards when the AUV goes close to the waypoint. Moreover, whereas r_{toward} was previously based on the difference $d_t - d_{t-1}$, it is here based only on the current distance d_t. The positive reward was before proportional to the speed at which the AUV is going towards the waypoint, whereas now it grows exponentially when the distance d_t is reduced. It better captures the goal of the task: we do not want the AUV to go to the waypoint as fast as possible, we simply want it to reach the waypoint independently of the duration.

6.3.2 Influence of the state vector's size on the performance

We will now see how the number of variables found in the state vector given by the environment to the SAC algorithm can affect its performance. We will start with the same state vector $\mathbf{S}_t$ as the one defined in the section 6.1.3.2, and we will remove variables until the agent is no more capable of learning the task. The task becomes harder to be fulfilled with fewer information. If the SAC is able to learn the task with less variables, this means that some sensors can be removed from the RexROV 2, which will allows to reduce the production costs of the AUV. We want to know the maximum number of variables we can remove from the state vector while still being able to fulfill the control task. Moreover, this sensitivity analysis of the state vector allows to identify which sensors are the more useful for the AUV.

6.3.2.1 Initial state vector

For this new task composed of closer waypoints, we first trained the SAC algorithm with the new changes described previously but with the same state vector $\mathbf{S}_t$ as for the task of the section 6.2:

$$\mathbf{S}_t = [\ \boldsymbol{x},\ \boldsymbol{\Theta},\ \boldsymbol{v},\ \boldsymbol{\omega},\ \theta_e,\ \psi_e,\ \boldsymbol{x}_e,\ \boldsymbol{u}_{t-1}\ ]^T \tag{6.13}$$

Let us recall that $\boldsymbol{x} = (x, y, z)$ is the position vector of the RexROV 2, $\boldsymbol{\Theta} = (\phi, \theta, \psi)$ is its orientation vector (the Euler angles), $\boldsymbol{v}$ is the linear velocity vector, $\boldsymbol{\omega}$ is the angular velocity vector, θ_e and ψ_e are respectively the tracking errors of the pitch and the yaw angles, $\boldsymbol{x}_e$ is the error between the position vector $\boldsymbol{x}$ and the position vector of the waypoint, and $\boldsymbol{u}_{t-1} = (u_1, u_2, u_3, u_4, u_5, u_6)$ is the action vector of the inputs sent at the previous time step. This vector is composed of 23 dimensions.

We kept the same procedure as in the previous trial. We trained the SAC algorithm for 2700 training episodes, and we had the following amount of success:

Number of success for each 100 episodes: [12, 60, 85, 95, 99, 99, 98, 99, 98, 100, 100, 100, 100, 100, 99, 100, 100, 100, 99, 99, 100, 100, 100, 100, 99, 100, 100]

The number of episodes after which we stopped the training phase will vary during the following results sections, since there is no rule of thumb for the number of training episodes needed for this task. Here the SAC managed to converge rapidly towards a good behaviour: it reaches 85% of success after only 300 episodes. The convergence was here slower than in the section 6.2, in which the SAC managed to have 91% during the first 100 episodes. This phenomenom can seem incoherent, since the harder task allowed a faster convergence of the SAC during the training phase. A possible explanation is that since the waypoints were placed further away from the initial position of the AUV (at most at 50 meters away on the X and Y axes), the episodes took more time steps to end, which led the replay buffer to be filled with more training samples at each episode than here. The update process of the networks parameters started therefore at earlier episodes, since this process begins when the replay buffer reaches a certain size.
During these trials,we started to introduce a new type of figure in order to monitor the training phase: the evolution of the cost functions of the SAC and of the total sum of rewards per episode. They are shown on the Figure 6.13 for the learning phase of this subsection. These plots were generated automatically during the training by matplotlib.
These plots describes the following features:

- The first light blue plot shows the evolution of the total reward per episode. The X axis represents the number of different training episodes and the Y axis describes the total sum of rewards obtained by the agent during each episode. It allows to see how fast the agent is able to find a behaviour allowing it to collect a high number of large positive rewards. However, a higher sum of rewards does not always mean a higher number of successes, since the agent can sometimes converge towards a sub-optimal behaviour leading to the highest possible sum of rewards at the expense of the success of the task. If such cases appear, the reward function needs to be fine-tuned in order to better reflect the goal of the task.

- The red plot shows the evolution of the Q-value loss function of the SAC. The X axis is composed of the time steps of the simulation and the Y axis describes the loss function J_Q used by the soft Q-value network (see the equation 3.41).

- The dark blue plot describes the evolution of the value loss function of the SAC. The X axis is composed of the time steps of the simulation and the Y axis describes the loss function J_V used by the soft value network (see the equation 3.40).

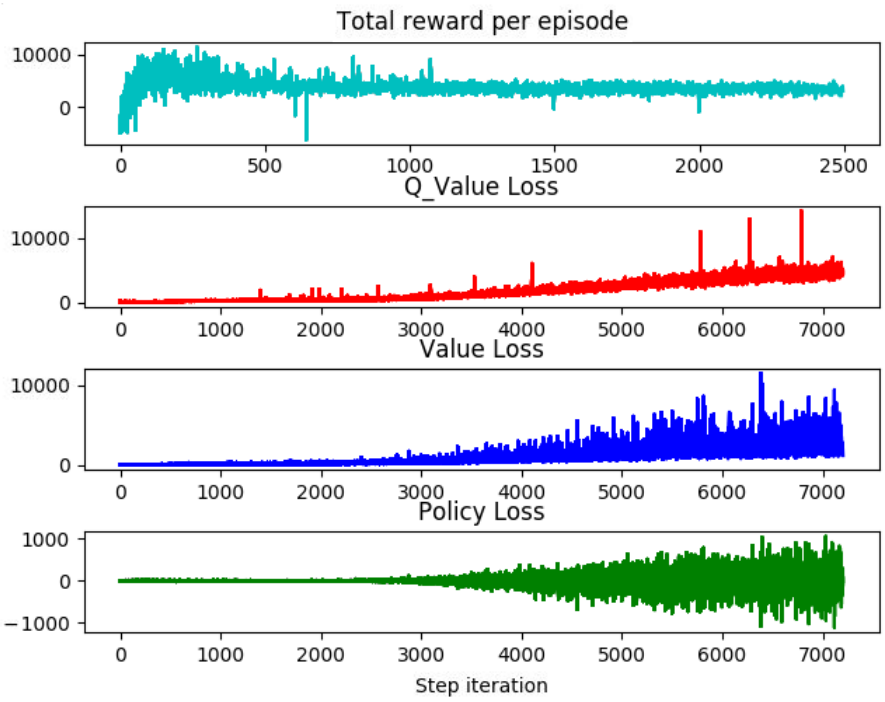


Figure 6.13. Total reward per episode and loss functions for the initial state vector.

- The green plot describes the evolution of the policy loss function of the SAC. The X axis is composed of the time steps of the simulation and the Y axis describes the loss function J_π used by the policy network (see the equation 3.45).
  The plot of the three loss functions allows to visualize when the learning process of the SAC algorithm happens. When a given loss function has a non-zero value for one time step, it means that the corresponding network does not manage to estimate correctly the value function, the Q-value function or the policy function: the outputs of the neural network differ from the targets (based on the training samples from the replay buffer). It means that its parameters need to be adjusted using the gradient descent. The higher the value of the loss function is, the more the parameters need to be adjusted.

Here the Figure 6.13 shows that the agent adopts rapidly a behaviour leading to very high rewards during the first 300 episodes, before converging towards an asymptote of slightly smaller total rewards per episode. The plots of the three loss functions also show that the agent continues to learn indefinitely, since the amplitude of the loss functions becomes bigger and bigger through time. The fact that these loss functions vary constantly from small values to high values is normal during the training phase:

- the high values are due to large errors between the expected outputs of the neural networks

and the real values (computed from the rewards sent by the environment). This means that new information needs to be learnt.

- the small values means that the observations of the environment match the expectations of the neural networks, when the agent goes from one state s_t to another state s_{t+1} thanks to the action a_t.

After the training process, we choose to test three different models in order to see how the number of training episodes performed by the agent can affect its performance during the testing phase. We choose to test a model after 600 training episodes, a model after 1300 training episodes and a model after 2500 training episodes. These three models were evaluated using the same procedure as the one described in the previous sections. In order to keep our simulations consistent, we will always use the three same models in the remaining results of the section 6.3.2. Here are the three results tables.

Metrics	PID	SAC
Success rate (%)	97.2	97.0
Collision rate (%)	2.8	3.0
Timeout failure rate (%)	0.0	0.0
Mean of $d\delta$ (m)	1.56	3.16
SD of $d\delta$ (m)	1.46	1.80
Mean of $\|\mathbf{u}\|$	531.31	413.36
Mean number of steps	200	209
Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.065	0.864
(Success) Mean of $d\delta$ (m)	1.42	2.95
(Success) SD of $d\delta$ (m)	1.27	1.57
(Success) Mean of $\|\mathbf{u}\|$	531.39	413.73
(Success) Mean number of steps	204	212
(Success) Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.081	0.874

Table 6.2. Testing phase of a model based on the initial state vector, trained on 600 episodes.

Metrics	PID	SAC
Success rate (%)	97.6	98.6
Collision rate (%)	2.2	1.4
Timeout failure rate (%)	0.2	0.0
Mean of $d\delta$ (m)	1.50	2.66
SD of $d\delta$ (m)	1.35	1.44
Mean of $\|\mathbf{u}\|$	531.63	446.00
Mean number of steps	202	175
Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.074	0.780
(Success) Mean of $d\delta$ (m)	1.37	2.64
(Success) SD of $d\delta$ (m)	1.17	1.41
(Success) Mean of $\|\mathbf{u}\|$	531.83	446.06
(Success) Mean number of steps	205	178
(Success) Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.085	0.790

Table 6.3. Testing phase of a model based on the initial state vector, trained on 1300 episodes.

Metrics	PID	SAC
Success rate (%)	97.0	98.2
Collision rate (%)	3.0	1.6
Timeout failure rate (%)	0.0	0.2
Mean of $d\delta$ (m)	1.73	2.95
SD of $d\delta$ (m)	1.63	2.11
Mean of $\|\mathbf{u}\|$	531.70	467.97
Mean number of steps	199	154
Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.057	0.720
(Success) Mean of $d\delta$ (m)	1.49	2.55
(Success) SD of $d\delta$ (m)	1.31	1.59
(Success) Mean of $\|\mathbf{u}\|$	531.69	469.55
(Success) Mean number of steps	203	155
(Success) Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.075	0.722

Table 6.4. Testing phase of a model based on the initial state vector, trained on 2500 episodes.

These tables are displaying very satifying results. First of all, the primary goal of the task is fulfilled: make the SAC have a success rate superior or equal to the PID controller. On this

run, the model 600 had the same sucess rate as the PID (within 0.2%), and the models 1300 and 2500 did slightly better (1% and 1.2% better respectively). Moreover, even if the PID stayed closer to the ideal trajectory (its mean and SD of $d\delta$ are always lower than ones of the SAC), the mean of $\sum \|\mathbf{u}\|$ was always lower than the PID. The SAC managed to save more energy than the PID controller, which was our secondary objective for this task. The models 1300 and 2500 also had slightly fewer collisions failures and took a smaller number of time steps to finish their test episodes, which is always good (even if it is not the focus of this work).
Unlike the model of the section 6.2, the SAC performed as well as the PID controller, and even better for some models. This seems logical since the task has been simplified in this section. However, we did not expect to have a better energy consumption while having such good success rates. **The SAC algorithm found a trade-off between fulfilling the control task and saving energy.**

6.3.2.2 Removing the measure of the position of the AUV

As of this section, we start to show the results of altering the state vector of the SAC algorithm. After removing the position vector $\boldsymbol{x} = (x, y, z)$ from the state vector, $\mathbf{S}_t$ has now 20 dimensions:

$$\mathbf{S}_t = [\ \boldsymbol{\Theta},\ \boldsymbol{v},\ \boldsymbol{\omega},\ \theta_e,\ \psi_e,\ \boldsymbol{x}_e,\ \boldsymbol{u}_{t-1}\ ]^T \tag{6.14}$$

After 3300 training episodes, the SAC had the following number of success during the training phase:

Number of success for each 100 episodes: [16, 70, 98, 95, 96, 98, 96, 99, 96, 95, 96, 96, 99, 99, 100, 97, 99, 98, 98, 98, 98, 98, 99, 96, 99, 98, 98, 100, 98, 97, 100, 98, 98]

We also obtained the cumulative rewards and the loss functions shown in the Figure 6.14. The total reward per episode is very similar to the previous one (in the Figure 6.13), but the amplitudes of the three loss functions are smaller, except at precise high peaks. These peaks could correspond to unexpected events appearing for the first time to the agent. Since these combinations of states, actions and rewards are new for it, it does not manage to predict these responses from the environment: the ouputs of the neural networks diverge from the targets, which generate these high peaks, because of the large Temporal Differential (TD) errors. We can also observe that the plot of the Q-value loss does not display a "ramp" shape as before.

As in the previous trials, the models selected after 600, 1300 and 2500 training episodes were tested and compared with the PID controller. Here are the results of these three test phases.

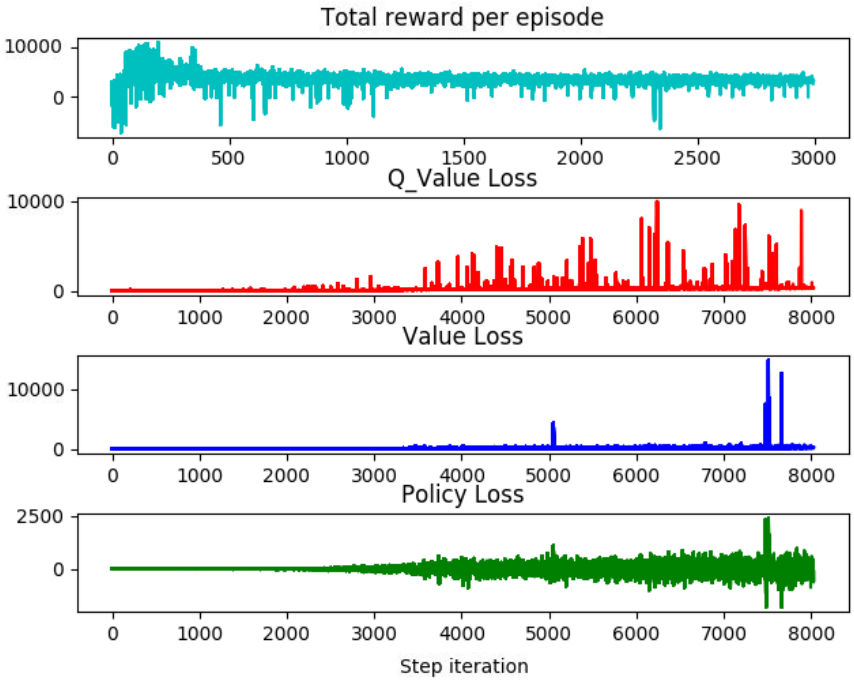


Figure 6.14. Total reward per episode and loss functions after removing the position from the state vector.

Metrics	PID	SAC
Success rate (%)	95.0	98.0
Collision rate (%)	4.8	1.4
Timeout failure rate (%)	0.2	0.6
Mean of $d\delta$ (m)	1.92	4.36
SD of $d\delta$ (m)	1.73	2.78
Mean of $\|\mathbf{u}\|$	533.45	427.21
Mean number of steps	205	240
Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.092	1.025
(Success) Mean of $d\delta$ (m)	1.45	4.08
(Success) SD of $d\delta$ (m)	1.13	2.48
(Success) Mean of $\|\mathbf{u}\|$	533.51	426.44
(Success) Mean number of steps	211	237
(Success) Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.121	1.005

Table 6.5. Testing phase of a model after removing the position from the state vector, trained on 600 episodes.

Metrics	PID	SAC
Success rate (%)	96.6	97.6
Collision rate (%)	3.4	1.8
Timeout failure rate (%)	0.0	0.6
Mean of $d\delta$ (m)	1.57	3.44
SD of $d\delta$ (m)	1.35	2.23
Mean of $\|\mathbf{u}\|$	532.44	462.14
Mean number of steps	201	180
Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.072	0.831
(Success) Mean of $d\delta$ (m)	1.45	3.09
(Success) SD of $d\delta$ (m)	1.19	1.82
(Success) Mean of $\|\mathbf{u}\|$	532.38	462.46
(Success) Mean number of steps	207	178
(Success) Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.097	0.817

Table 6.6. Testing phase of a model after removing the position from the state vector, trained on 1300 episodes.

Metrics	PID	SAC
Success rate (%)	97.4	98.2
Collision rate (%)	2.4	1.8
Timeout failure rate (%)	0.2	0.0
Mean of $d\delta$ (m)	1.76	3.26
SD of $d\delta$ (m)	1.57	2.11
Mean of $\|\mathbf{u}\|$	532.85	479.79
Mean number of steps	205	161
Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.094	0.772
(Success) Mean of $d\delta$ (m)	1.48	2.96
(Success) SD of $d\delta$ (m)	1.24	1.76
(Success) Mean of $\|\mathbf{u}\|$	532.47	479.75
(Success) Mean number of steps	205	159
(Success) Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.088	0.760

Table 6.7. Testing phase of a model after removing the position from the state vector, trained on 2500 episodes.

The three models were all able to have a slightly better success rate than the PID controller

and with a fewer percentage of collisions, which is better than before. As before, the PID always had a smaller distance error $d\delta$, but **the SAC always manage to save more energy by having a smaller mean of $\sum \|\mathbf{u}\|$.**
Thanks to this trial, we know that the agent of **the SAC algorithm can control the RexROV 2 without knowing the true global position vector. However, it still needs to know its position relatively to the waypoint, given by the error position vector $\boldsymbol{x}_e$.**

6.3.2.3 Removing the measure of the Euler angles and the pitch tracking error

After removing the position vector of the AUV from the state vector of the SAC, we now remove the measure of the Euler angles, given by the orientation vector $\boldsymbol{\Theta} = (\phi, \theta, \psi)$, and the pitch tracking error θ_e (the amount of pitch angle that is lacking in order to make the AUV point towards the waypoint). $\mathbf{S}_t$ becomes a 16-dimensional vector:

$$\mathbf{S}_t = [\ \boldsymbol{v},\ \boldsymbol{\omega},\ \psi_e,\ \boldsymbol{x}_e,\ \boldsymbol{u}_{t-1}\ ]^T \tag{6.15}$$

After 3400 training episodes, the SAC had the following number of success during the training phase:

Number of success for each 100 episodes: [43, 100, 98, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 99, 99, 100, 100, 100, 99, 100, 100, 100, 100, 99, 100, 100, 99, 100, 99, 100, 100, 100, 100]

The Figure 6.15 shows the cumulative rewards and the loss functions we get during the training phase. All the four plots are similar to the plots found in the Figure 6.13, but are much noisier. Perhaps some situations encountered during the training phase were more extreme than during the previous training phases, troubling the learning process of the agent at some time steps.

Like previously, the models selected after 600, 1300 and 2500 training episodes were tested and compared with the PID controller. Here are the results of these three test phases.

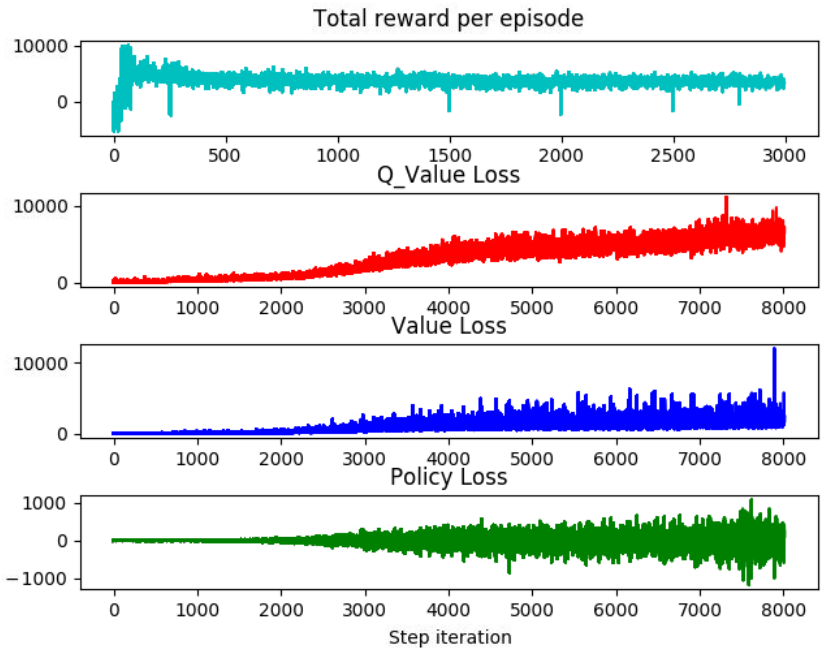


Figure 6.15. Total reward per episode and loss functions after removing the Euler angles and the pitch error from the state vector.

Metrics	PID	SAC
Success rate (%)	97.0	97.6
Collision rate (%)	2.8	2.4
Timeout failure rate (%)	0.2	0.0
Mean of $d\delta$ (m)	1.85	4.27
SD of $d\delta$ (m)	1.76	2.59
Mean of $\|\mathbf{u}\|$	532.72	429.72
Mean number of steps	209	239
Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.113	1.027
(Success) Mean of $d\delta$ (m)	1.54	4.21
(Success) SD of $d\delta$ (m)	1.35	2.51
(Success) Mean of $\|\mathbf{u}\|$	532.66	429.52
(Success) Mean number of steps	214	242
(Success) Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.134	1.034

Table 6.8. Testing phase of a model after removing the Euler angles and the pitch error from the state vector, trained on 600 episodes.

Metrics	PID	SAC
Success rate (%)	97.8	99.4
Collision rate (%)	2.2	0.6
Timeout failure rate (%)	0.0	0.0
Mean of $d\delta$ (m)	1.50	3.16
SD of $d\delta$ (m)	1.30	1.70
Mean of $\|\mathbf{u}\|$	533.04	467.87
Mean number of steps	203	189
Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.081	0.883
(Success) Mean of $d\delta$ (m)	1.33	3.15
(Success) SD of $d\delta$ (m)	1.08	1.69
(Success) Mean of $\|\mathbf{u}\|$	533.12	467.91
(Success) Mean number of steps	206	190
(Success) Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.092	0.887

Table 6.9. Testing phase of a model after removing the Euler angles and the pitch error from the state vector, trained on 1300 episodes.

Metrics	PID	SAC
Success rate (%)	96.2	98.8
Collision rate (%)	3.8	1.2
Timeout failure rate (%)	0.0	0.0
Mean of $d\delta$ (m)	1.45	4.96
SD of $d\delta$ (m)	1.40	2.74
Mean of $\|\mathbf{u}\|$	532.82	457.09
Mean number of steps	199	336
Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.058	1.536
(Success) Mean of $d\delta$ (m)	1.30	4.86
(Success) SD of $d\delta$ (m)	1.19	2.65
(Success) Mean of $\|\mathbf{u}\|$	532.82	457.26
(Success) Mean number of steps	206	335
(Success) Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.093	1.529

Table 6.10. Testing phase of a model after removing the Euler angles and the pitch error from the state vector, trained on 2500 episodes.

The three models had a better success rate and a better collision rate than the PID con-

troller, and **the model trained on 1300 episodes even achieved a sucess rate of 99.4%, meaning that it did not reached the waypoint during only 3 test episodes.** Like previously, the PID controller had a better mean and a better SD for the distance error $d\delta$. The models 600 and 1300 managed to save more energy than the PID, since they had a lower mean $\sum \|\mathbf{u}\|$, but for the first time the PID managed to beat the model 2500 on this criterion (on all episodes and on successful episodes).
These results show that **the SAC algorithm does not need to know neither its orientation vector $\boldsymbol{\Theta}$ nor the pitch tracking error θ_e in order to perform the task. The only angle it needs to know is the yaw tracking error ψ_e.**

6.3.2.4 Removing the measure of the angular speeds

For this trial, we removed the angular velocity vector $\boldsymbol{\omega}$ from $\mathbf{S}_t$, leaving the state vector with 13 dimensions:

$$\mathbf{S}_t = [\ \boldsymbol{v},\ \psi_e,\ \boldsymbol{x}_e,\ \boldsymbol{u}_{t-1}\ ]^T \tag{6.16}$$

After 4000 training episodes, the SAC had the following number of success during the training phase:

Number of success for each 100 episodes: [51, 99, 98, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 99, 100, 100, 100, 100, 99, 100, 100, 100, 100, 99, 100, 100, 100, 100, 99, 100, 100, 100, 100, 99]

The Figure 6.16 shows the cumulative rewards and the loss functions obtained during the training phase. The four plots are very similar to the previous plots.

Like previously, the models selected after 600, 1300 and 2500 training episodes were tested and compared with the PID controller. Here are the results of these three test phases.

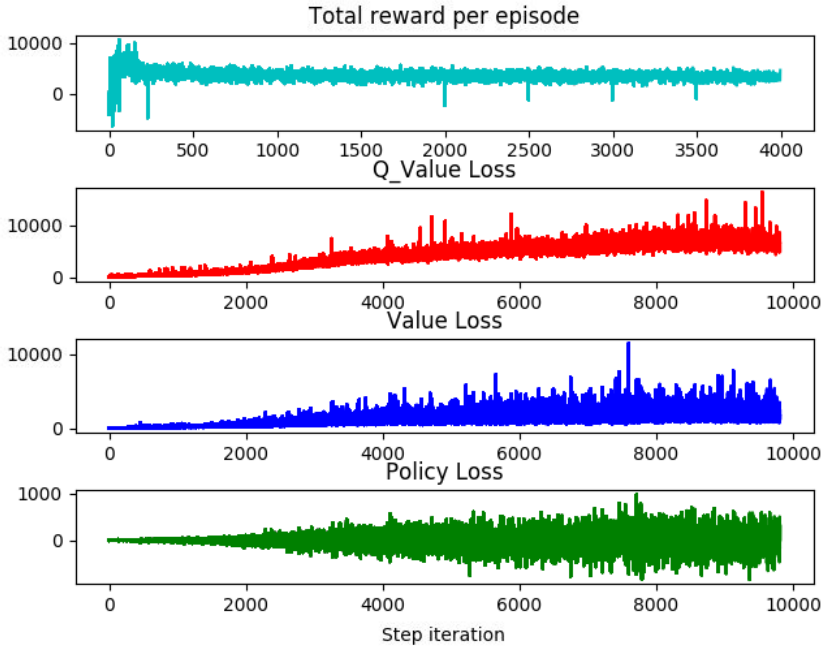


Figure 6.16. Total reward per episode and loss functions after removing the angular speeds from the state vector.

Metrics	PID	SAC
Success rate (%)	99.8	96.8
Collision rate (%)	0.0	3.0
Timeout failure rate (%)	0.2	0.2
Mean of $d\delta$ (m)	1.40	4.63
SD of $d\delta$ (m)	1.27	3.17
Mean of $\|\mathbf{u}\|$	533.58	428.44
Mean number of steps	208	260
Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.109	1.113
(Success) Mean of $d\delta$ (m)	1.32	3.72
(Success) SD of $d\delta$ (m)	1.17	2.14
(Success) Mean of $\|\mathbf{u}\|$	533.18	428.55
(Success) Mean number of steps	207	251
(Success) Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.100	1.071

Table 6.11. Testing phase of a model after removing the angular speeds from the state vector, trained on 600 episodes.

Metrics	PID	SAC
Success rate (%)	96.2	97.4
Collision rate (%)	3.4	2.4
Timeout failure rate (%)	0.4	0.2
Mean of $d\delta$ (m)	1.71	4.25
SD of $d\delta$ (m)	1.74	2.89
Mean of $\|\mathbf{u}\|$	532.84	448.30
Mean number of steps	202	231
Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.074	1.034
(Success) Mean of $d\delta$ (m)	1.24	3.93
(Success) SD of $d\delta$ (m)	1.13	2.52
(Success) Mean of $\|\mathbf{u}\|$	532.63	447.45
(Success) Mean number of steps	204	231
(Success) Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.080	1.028

Table 6.12. Testing phase of a model after removing the angular speeds from the state vector, trained on 1300 episodes.

Metrics	PID	SAC
Success rate (%)	98.4	98.2
Collision rate (%)	1.4	1.8
Timeout failure rate (%)	0.2	0.0
Mean of $d\delta$ (m)	1.57	2.72
SD of $d\delta$ (m)	1.26	1.52
Mean of $\|\mathbf{u}\|$	533.22	468.51
Mean number of steps	206	159
Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.101	0.743
(Success) Mean of $d\delta$ (m)	1.50	2.67
(Success) SD of $d\delta$ (m)	1.18	1.46
(Success) Mean of $\|\mathbf{u}\|$	533.01	468.66
(Success) Mean number of steps	208	161
(Success) Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.104	0.752

Table 6.13. Testing phase of a model after removing the angular speeds from the state vector, trained on 2500 episodes.

For the models 600 and 2500, the PID controller had a better success rate and a better

collision rate than the SAC-based controller, whereas it was the contrary for the model 1300. This means that the model 600 has not learnt enough information, and that the model 2500 has been trained too long and is probably biased: the model 2500 tried to maximize the cumulative rewards at all cost, even if the episodes ended with a failure. For the three models, the PID controller still had a better mean and a better SD for distance error $d\delta$. **For the models 1300 and 2500, the SAC saved more energy by having a lower mean $\sum \|\mathbf{u}\|$.** For the model 600, we see that the PID controller had a lower mean $\sum \|\mathbf{u}\|$ when we take all the episodes into account (the general *Mean of* $\sum \|\mathbf{u}\|$ metric), but it is the contrary when we only take the successful episodes into account (*(Success) Mean of* $\sum \|\mathbf{u}\|$ metric). This shows us once again that **taking into account all the test episodes can biased the interpretation of the results, especially if we want to only analyse the manner the controllers reach the target waypoints.**
These results show that **the SAC algorithm does not need to know any information about its angular speeds in order to reach the waypoints.**

6.3.2.5 Replacing the position tracking errors with the relative distance to the waypoint

For this run, we replaced the position tracking vector $\boldsymbol{x}_e$ (the error between the position vector $\boldsymbol{x}$ and the position vector of the waypoint) by d_t, the scalar giving the relative distance between the AUV and the target waypoint. $\mathbf{S}_t$ becomes a 11-dimensional vector:

$$\mathbf{S}_t = [\ \boldsymbol{v},\ \psi_e,\ d_t,\ \boldsymbol{u}_{t-1}\ ]^T \tag{6.17}$$

After 2400 training episodes, the SAC had the following number of success during the training phase:

Number of success for each 100 episodes: [15, 33, 33, 31, 36, 8, 19, 1, 13, 21, 17, 1, 7, 2, 18, 2, 3, 10, 39, 40, 36, 22, 14, 26]

The SAC did not managed to converge towards a good behaviour with this state vector. The agent is lacking of information and did not managed to understand neither the goal of the task, nor the dynamics of the RexROV 2. We did not lost time to compare any model with the PID controller, since every trial or simulation takes hours or days to be completed.

Finally we can note that the plots shown on the Figure 6.17 are noisier in the case of a failure in the learning process. The total reward per episode also did not converge towards an asymptote since the agent did not managed to find high cumulative rewards.

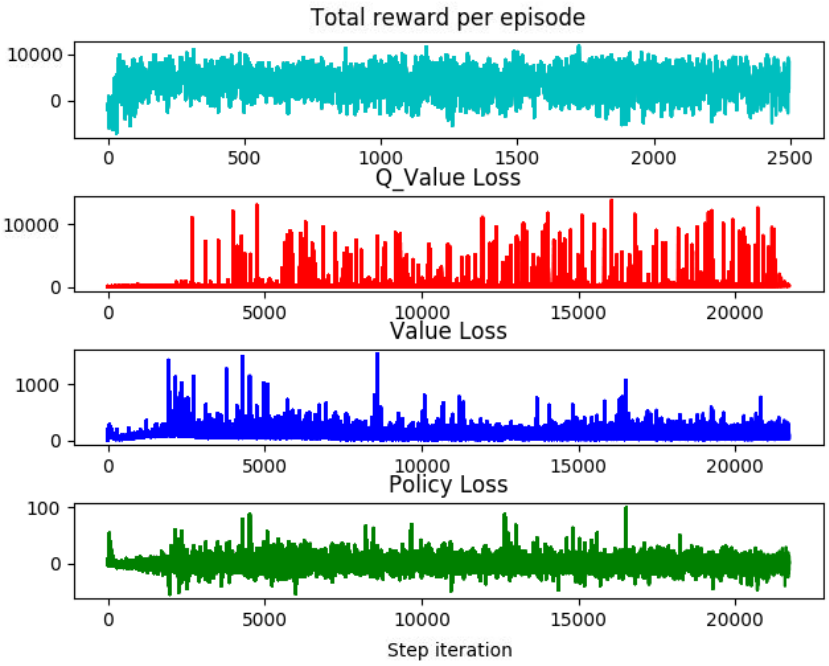


Figure 6.17. Total reward per episode and loss functions after replacing the position errors with the relative distance to the waypoint.

6.3.2.6 Removing the measure of the linear speeds

After failing to make the SAC algorithm learn the control task in the previous trial, we restored the position tracking vector $\boldsymbol{x}_e$ in the vector $\mathbf{S}_t$ (instead of d_t), and we removed the linear velocity vector $\boldsymbol{v}$. The vector $\mathbf{S}_t$ has now 10 dimensions:

$$\mathbf{S}_t = [\ \psi_e,\ \boldsymbol{x}_e,\ \boldsymbol{u}_{t-1}\ ]^T \tag{6.18}$$

After 2500 training episodes, the SAC had the following number of success during the training phase:

Number of success for each 100 episodes: [1, 26, 48, 76, 78, 87, 89, 90, 72, 77, 81, 85, 89, 90, 91, 95, 96, 96, 62, 68, 24, 47, 45, 61, 46]

The SAC never reached a success rate of 100% during this training phase, but it still managed to have more than 95% of success rate several time. Moreover, the usual training plots shown on the Figure 6.18 appears to be less noisy than before.

As of this subsection, **less models will be tested in order to save computing time**: during this work, we had a lot of simulations to run in order to test a large range of config-

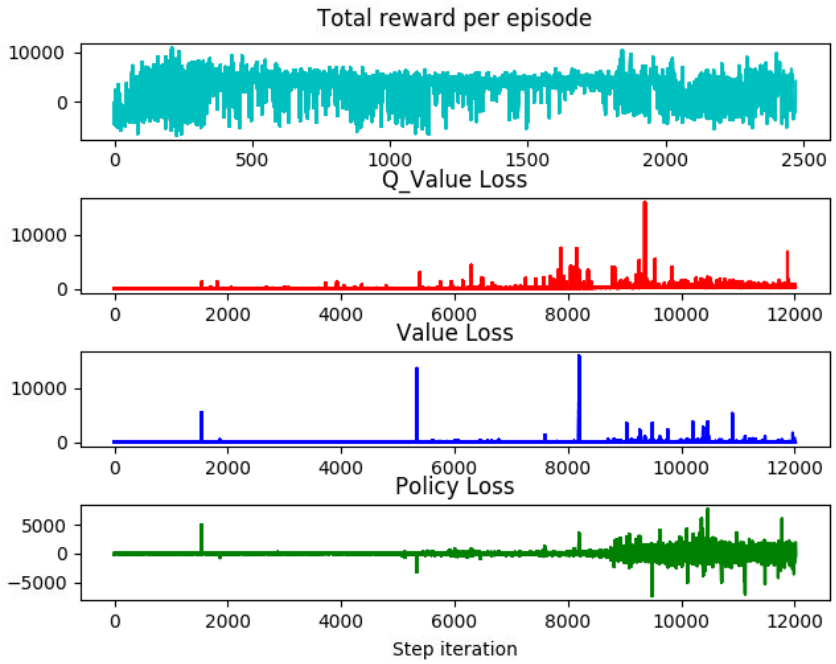


Figure 6.18. Total reward per episode and loss functions after removing the linear speeds from the state vector.

uration and implementations of the SAC algorithm. Moreover, we saw previously than even if the models were trained on different amount of episodes, the final performance of all these tested models were quite similar (as long as the SAC algorithm managed to converge to a good behaviour during the learning process). Testing less models is still representative of the abilities of the SAC algorithm.
For this trial, we only tested the model 1750. This model was chosen by checking the success rates obtained during the learning process: the SAC had 96% of success rate during 200 successive episodes, between the episode 1601 and the episode 1800. Here is the corresponding results table.

Metrics	PID	SAC
Success rate (%)	99.2	97.6
Collision rate (%)	0.2	0.8
Timeout failure rate (%)	0.6	1.6
Mean of $d\delta$ (m)	1.63	6.54
SD of $d\delta$ (m)	1.39	4.93
Mean of $\|\mathbf{u}\|$	532.62	459.05
Mean number of steps	208	248
Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.109	1.137
(Success) Mean of $d\delta$ (m)	1.45	4.84
(Success) SD of $d\delta$ (m)	1.20	2.90
(Success) Mean of $\|\mathbf{u}\|$	532.31	461.29
(Success) Mean number of steps	205	234
(Success) Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.085	1.077

Table 6.14. Testing phase of a model after removing the linear speeds from the state vector, trained on 1750 episodes.

The model 1750 reached 97.6% of success rate. Even if the PID did slightly better (with a difference of 1.6% between the two success rates), it is still an excellent performance from the SAC algorithm. The success rate did not reach the symbolic 100% barrier during this training phase, but the task was as good performed as during the previous testing phases. **The PID controller saved more energy than the SAC on all episodes, but it is the contrary if we take into account only the successful episodes.** For us, the mean of $\sum \|\mathbf{u}\|$ computed on the successful episodes is more meaningful than when it is computed on all the episodes: it allows to better compare the tracking ability of the controllers, without being biased by the episodes where the AUV goes outside of the box, or stagnates without reaching the target waypoint. We can also note that **the SAC had surprisingly better success rates during the testing phase than during the training phase**, where it reached 96% at most.
The SAC algorithm is still able to learn the waypoint tracking task and to understand the AUV dynamics without any velocities information (neither angular nor linear).

6.3.2.7 Removing the measure of the yaw tracking error

For this trial, we removed the yaw tracking error ψ_e from $\mathbf{S}_t$, leaving the state vector with only 9 dimensions:

$$\mathbf{S}_t = [\ \boldsymbol{x}_e,\ \boldsymbol{u}_{t-1}\ ]^T \tag{6.19}$$

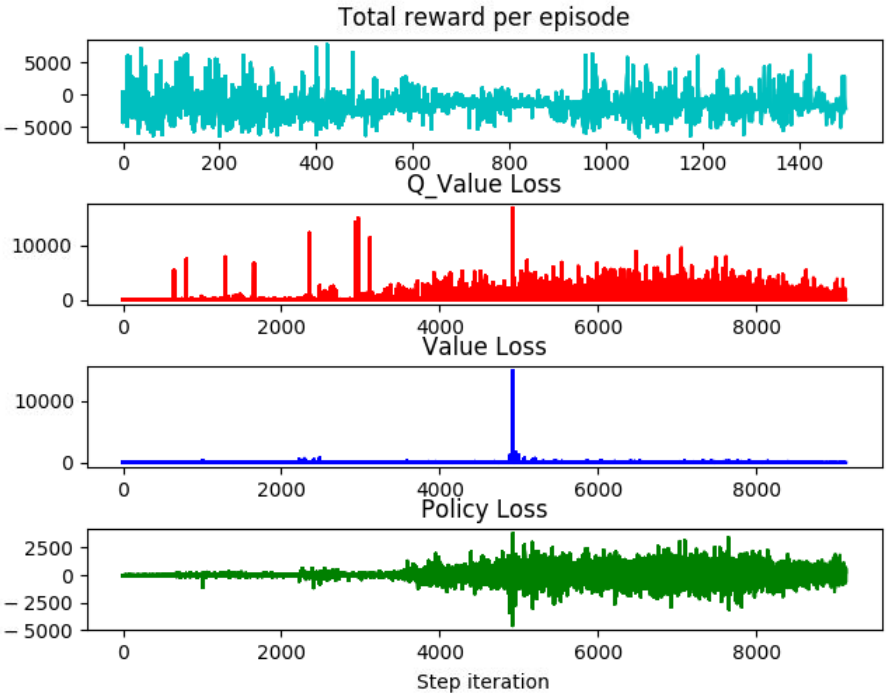


Figure 6.19. Total reward per episode and loss functions after removing the yaw error from the state vector.

After 1500 training episodes, the SAC had the following number of success during the training phase:

Number of success for each 100 episodes: [3, 4, 2, 0, 2, 0, 0, 0, 0, 1, 2, 2, 0, 3, 1]

The SAC algorithm did not manage to converge towards a good behaviour. The total reward per episode plot found in the Figure 6.19 confirms that the learning process failed: the curve is most of the time in the negative values. **This failure is probably due to the lack of information given by the environment within the state vector.**

Considering the extremely low values of the success rates obtained during the learning phase, we did not test any models for this trial.

6.3.2.8 Removing the values of the previous inputs

After the failure of the previous trial, we restored the yaw tracking error ψ_e and removed the vector of the past inputs $\boldsymbol{u}_{t-1}$ from $\mathbf{S}_t$, leaving the state vector with only 4 dimensions:

$$\mathbf{S}_t = [\ \psi_e,\ \boldsymbol{x}_e\ ]^T \tag{6.20}$$

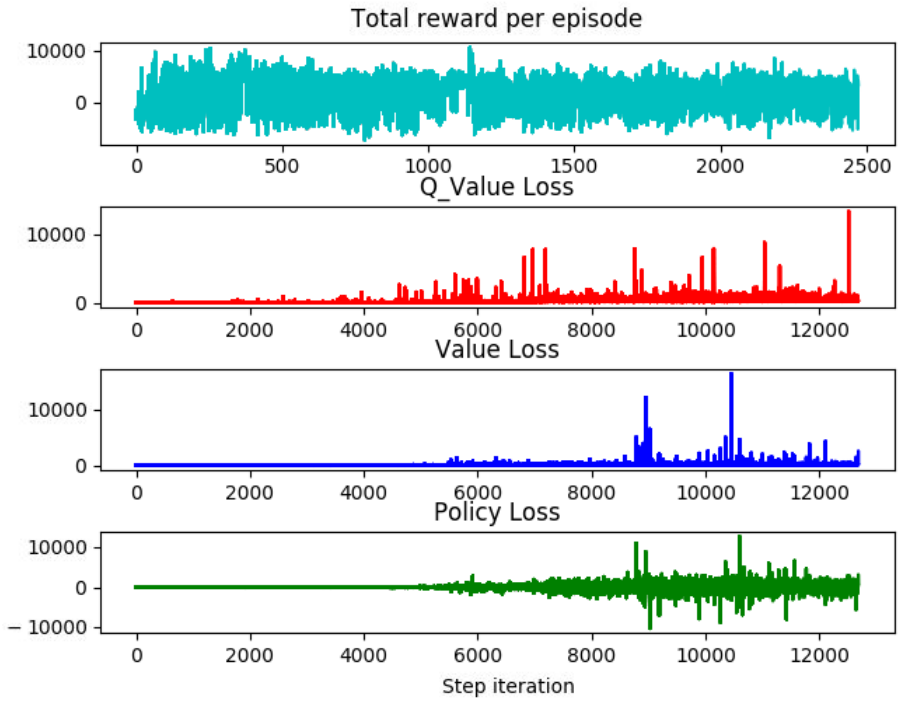


Figure 6.20. Total reward per episode and loss functions after removing the past inputs from the state vector.

After 2500 training episodes, the SAC had the following number of success during the training phase:

Number of success for each 100 episodes: [6, 16, 45, 67, 65, 70, 70, 65, 68, 68, 71, 77, 52, 53, 54, 58, 45, 45, 57, 36, 53, 73, 63, 63, 46]

The learning process did not managed to get more than 80%, so we cannot say that the SAC's agent converged towards a satisfactory behaviour. **However it still reached more than 75% and it is still worth to compare it with the PID controller.** We can also see on the Figure 6.20 that the total reward per episode has a large variance, since the plot spent almost an equal amount of time steps inside the positive values and the negative values. **This confirms that this state vector configuration made the agent struggle during the training phase.**

We chose to test the model 1100: during the training phase, the success rate reached 71% before the episode 1100 and 77% after it. Here is the corresponding results table.

Metrics	PID	SAC
Success rate (%)	97.8	75.4
Collision rate (%)	2.0	16.4
Timeout failure rate (%)	0.2	8.2
Mean of $d\delta$ (m)	2.42	6.36
SD of $d\delta$ (m)	2.53	3.94
Mean of $\|\mathbf{u}\|$	532.78	427.06
Mean number of steps	202	354
Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.075	1.512
(Success) Mean of $d\delta$ (m)	1.50	4.81
(Success) SD of $d\delta$ (m)	1.23	2.74
(Success) Mean of $\|\mathbf{u}\|$	532.74	430.42
(Success) Mean number of steps	204	295
(Success) Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	1.082	1.265

Table 6.15. Testing phase of a model after removing the linear speeds from the past inputs, trained on 1100 episodes.

Compared to results of the section 6.3.2.6 (where the SAC obtained a success rate of 97.6%), **removing the past inputs from the state vector made the success drops to 75.4%.** The collision and timeout failure rates strongly increased, which further confirms the drop in performance. Moreover **the SAC-based controller was not able to save more energy than the PID controller, making it worst than the PID on all the criteria.**

Even if the PID did better on this trial, the SAC algorithm still managed to a sub-optimal behaviour with a state vector of only 4 dimensions. A success rate of 75.4% is still better than the results if the sections 6.3.2.5 and 6.3.2.7, where the SAC did not managed to converge at all, despite having a larger state vector.

6.3.2.9 Replacing the position tracking errors with the pitch error

After failing to make the SAC algorithm learn the control task in the previous trial, we restored the vector of the past inputs $\boldsymbol{u}_{t-1}$ in the vector $\mathbf{S}_t$, and we replaced the position tracking vector $\boldsymbol{x}_e$ with the pitch tracking error θ_e. The vector $\mathbf{S}_t$ has now 8 dimensions:

$$\mathbf{S}_t = [\ \theta_e,\ \psi_e,\ \boldsymbol{u}_{t-1}\ ]^T \tag{6.21}$$

After 800 training episodes, the SAC had the following number of success during the training phase:

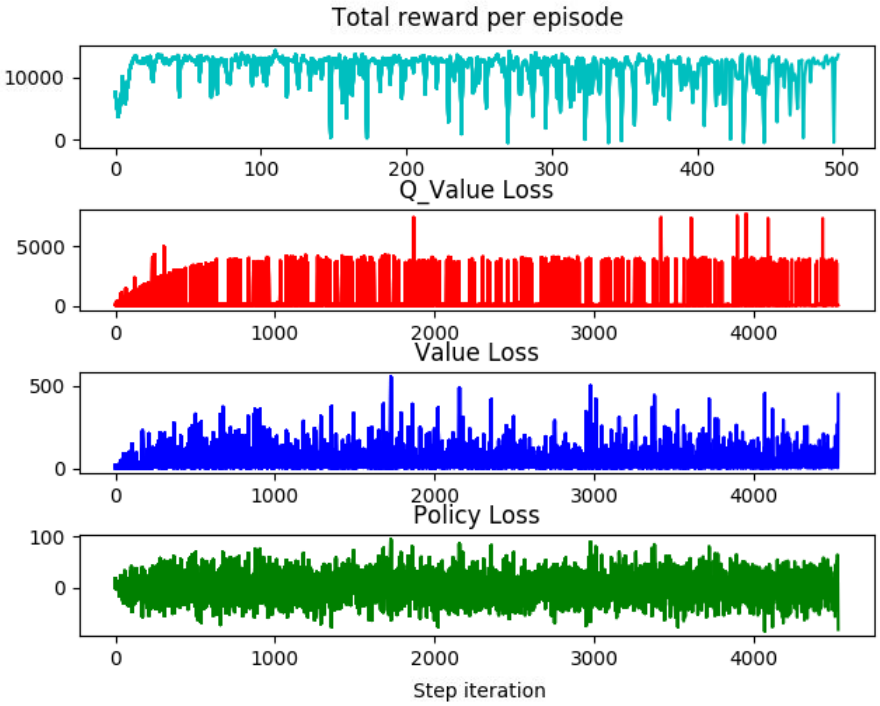


Figure 6.21. Total reward per episode and loss functions after replacing the position errors with the pitch error.

Number of success for each 100 episodes: [1, 0, 1, 2, 2, 0, 0, 0]

The SAC algorithm did not manage to learn the control task and the dynamics of the RexROV 2. The 6 successes obtained during the first 500 training episodes did not give enough information to the SAC to converge, which is due to the fact that the state vector do not contain enough useful variables. The Figure 6.21 was generated after 500 episodes and do not seem different from the previous training plots, even if the learning process was catastrophic here.

We did not run testing phases, because of the too low amount of successes obtained during the training phase. Moreover, we did not test other state vector configurations, because we judged that the vectors from 6.3.2.6 and 6.3.2.8 were the minimum acceptable configurations we implemented.

6.3.3 Sum up of the results on the simpler task

Thanks to this sensitivity analysis of the size of the sate vector, the SAC algorithm managed to fulfill this simplified task with a reduced state vector $\mathbf{S}_t$. **In the section 6.3.2.6, the state**

vector of the SAC algorithm was set to its smallest size while still allowing the controller to perform as good as the PID controller. The state vector was reduced from 23 to 10 dimensions and was composed of the following variables:

$$\mathbf{S}_t = [\ \psi_e,\ \boldsymbol{x}_e,\ \boldsymbol{u}_{t-1}\ ]^T \tag{6.22}$$

In almost all cases until this configuration, the SAC algorithm managed to equalize the success rate of the PID controller, and very often it had a slightly better one. Even if the PID controller was always closer to ideal trajectory, the SAC almost always saved more energy than the PID, by generating lower cumulative inputs during the test episodes. It could allow to perform longer missions.

In the section 6.3.2.8, the SAC managed to converge towards a sub-optimal behaviour despite having a state vector with only 4 dimensions, which shows its learning abilities. During the testing phase, the PID controller outperformed the SAC on all aspects, but the SAC-based controller still reached a success rate of 75.4% with the following state vector:

$$\mathbf{S}_t = [\ \psi_e,\ \boldsymbol{x}_e\ ]^T \tag{6.23}$$

The best success rate was obtained by the model 1300 tested in the section 6.3.2.3. The state vector of the SAC algorithm was then composed of 16 dimensions:

$$\mathbf{S}_t = [\ \boldsymbol{v},\ \boldsymbol{\omega},\ \psi_e,\ \boldsymbol{x}_e,\ \boldsymbol{u}_{t-1}\ ]^T \tag{6.24}$$

The best energy saving performance is assigned to the model achieving the lowest mean of $\sum \|\mathbf{u}\|$. It corresponds to the model 2500 of the section 6.3.2.1, trained with the initial 23-dimensional state vector:

$$\mathbf{S}_t = [\ \boldsymbol{x},\ \boldsymbol{\Theta},\ \boldsymbol{v},\ \boldsymbol{\omega},\ \theta_e,\ \psi_e,\ \boldsymbol{x}_e,\ \boldsymbol{u}_{t-1}\ ]^T \tag{6.25}$$

Based on all these results, we can also note that **the number of training episodes does not influence the performance after a certain threshold.** Once the SAC algorithm converged towards a satisfactory behaviour, training on more episodes did not improve the success rates of the SAC-based controller. This threshold can be found in the array of the *Number of success for each 100 episodes* metric: once the training reach more than 70% of success several times in a row, we can consider that the model have been sufficiently trained. If the performance of a given model is not suitable or if we want to see that a model can have better results, we can try another model, by selecting it among the ones beyond the threshold. This process of selecting the right model is rather empirical and task-dependant.

Finally, we saw that **the training plots (composed of the total reward per episode and the three loss functions of the neural networks of the SAC) do not help in**

supervising the training process better than with just having the *Number of success for each 100 episodes*. The figures remained quite similar during all the training phases, even when the SAC algorithm did not manage to converge towards a satisfactory behaviour like in the section 6.3.2.5.

6.4 Improving the performances on a harder task using advanced training techniques

After successfully making the SAC algorithm learn to perform the simpler task (the task with target waypoints placed closer from the initial target waypoint), we are going to try to improve the performance on the harder task. We want to see if the use of advanced techniques can improve the training of the SAC algorithm when it is struggling to learn a waypoint tracking task.
From now, all the trials will be done on the same task as in the section 6.2 (the harder task): the waypoints will be randomly placed inside a large box with the ranges [-50: 50] on the X and Y axes and [-60: -1] on the Z axis. The *collision* failure events will only happen when the AUV is leaving the boundaries [-60: -1] (the collisions events on the X and Y axes has been removed, compared to the previous section), since we do not want to add additional difficulty to this long distance waypoint tracking task.
For all the next trials, we will use the initial state vector taken from the subsection 6.3.2.1:

$$\mathbf{S}_t = [\ \boldsymbol{x},\ \boldsymbol{\Theta},\ \boldsymbol{v},\ \boldsymbol{\omega},\ \theta_e,\ \psi_e,\ \boldsymbol{x}_e,\ \boldsymbol{u}_{t-1}\ ]^T \tag{6.26}$$

where $\boldsymbol{x} = (x, y, z)$ is the position vector of the RexROV 2, $\boldsymbol{\Theta} = (\phi, \theta, \psi)$ is its orientation vector (the Euler angles), $\boldsymbol{v}$ is the linear velocity vector, $\boldsymbol{\omega}$ is the angular velocity vector, θ_e and ψ_e are respectively the tracking errors of the pitch and the yaw angles, $\boldsymbol{x}_e$ is the error between the position vector $\boldsymbol{x}$ and the position vector of the waypoint, and $\boldsymbol{u}_{t-1} = (u_1, u_2, u_3, u_4, u_5, u_6)$ is the action vector of the inputs sent at the previous time step. This vector is composed of 23 dimensions. We are keeping the same state vector configuration as during our first trial in the section 6.2 in order to not complicate too much this harder task (less components means less information for understanding the task and the AUV dynamics). This section is independent from the sensitivity analysis of the state vector performed in the section 6.3.
More, we kept some changes made to the simulation setup in the section 6.3.1. The simulation is kept in real-time configuration in order to better transfer the SAC algorithm to embedded systems. We are also keeping the following reward function structure (we replaced each constant by its numerical value for more clarity):

$$r_t = \begin{cases} 500 & \textbf{if } d_t < 3 \text{ (success)} \\ -550 & \textbf{if } z \notin [-60 : -1] \text{ (collision)} \\ 40. \exp\left(-\frac{d_t}{20}\right) & \textbf{if } d_t < d_{t-1} \\ -10 & \textbf{if } d_t \geq d_{t-1} \end{cases} \tag{6.27}$$

where d_t is the distance between the position of the AUV and the waypoint at the time step t.

In the following subsections, we propose to improve the learning process of this harder task by using advanced training techniques.

6.4.1 Initial trial on the harder task

We first trained normally the SAC algorithm on this new harder task. We kept all the changes described previously and we followed the same procedure as in the previous sections.
After 2500 training episodes, the SAC had the following number of success during the training phase:

Number of success for each 100 episodes: [0, 8, 21, 29, 44, 49, 62, 66, 83, 54, 37, 71, 70, 74, 79, 60, 75, 67, 52, 20, 8, 51, 20, 51, 52]

We can see that the SAC algorithm took much more time to converge towards a sufficient behaviour since it passed the 50% of success after 700 training episodes, while this threshold was reach during the 200 first episodes in the previous section. Moreover, **it did not manage to reach more than 90% of success rate during the whole training phase.**

The Figure 6.22 shows the cumulative rewards and the loss functions obtained during the training phase. The three plots of the loss functions are very similar to those found in the Figure 6.3.2.2, which means that the neural networks are learning new information throughout the training process. However, the cumulative rewards plot looks very different from the previous results, and did not converge towards an asymptote. The sum of rewards per episode often fall in the negative values, meaning that **the training is very unstable**, leading the agent of the SAC algorithm to not correctly maximizing the return 6.7. It is perhaps exploring too much its environment instead of seeking the best sum of rewards.

Since the model did not reach more than 80% before 900 episodes during the training phase, we only selected two models for the testing phase: the models trained on 1300 and 2500 episodes. We followed the same testing method as for the simpler task, but with long distance waypoints. Here are the two results tables.

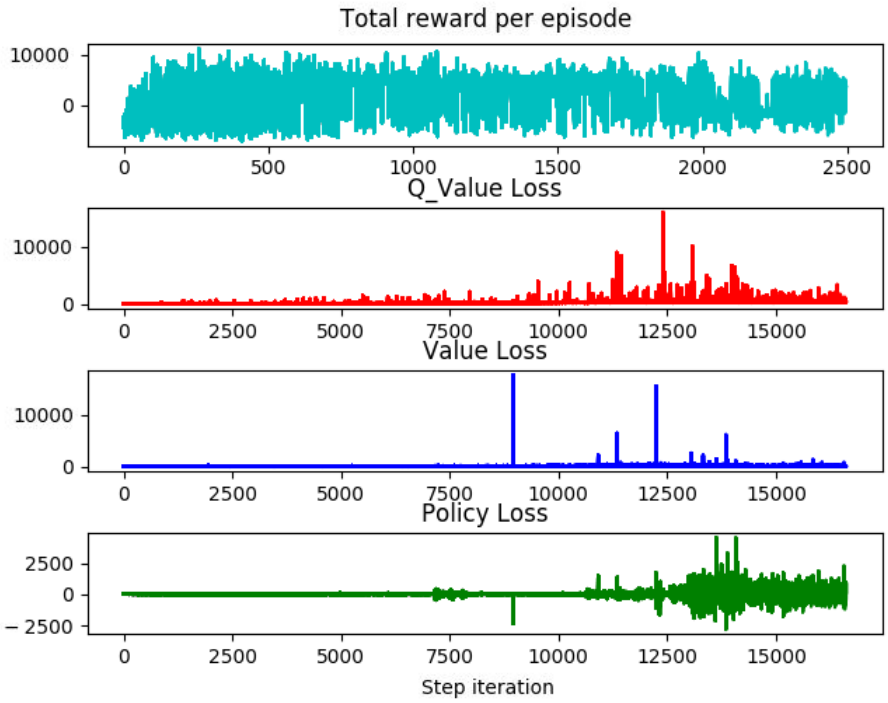


Figure 6.22. Total reward per episode and loss functions for a normal training on long distance waypoints.

Metrics	PID	SAC
Success rate (%)	93.8	76.0
Collision rate (%)	4.4	10.2
Timeout failure rate (%)	1.8	13.8
Mean of $d\delta$ (m)	1.71	8.14
SD of $d\delta$ (m)	1.95	5.60
Mean of $\|\mathbf{u}\|$	529.14	393.19
Mean number of steps	406	503
Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	2.151	1.977
(Success) Mean of $d\delta$ (m)	1.25	5.46
(Success) SD of $d\delta$ (m)	1.37	3.26
(Success) Mean of $\|\mathbf{u}\|$	528.59	404.05
(Success) Mean number of steps	405	421
(Success) Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	2.134	1.698

Table 6.16. Testing phase of a model trained normally on long distance waypoints for 1300 episodes.

Metrics	PID	SAC
Success rate (%)	94.6	35.2
Collision rate (%)	4.2	58.2
Timeout failure rate (%)	1.2	6.6
Mean of $d\delta$ (m)	2.52	9.36
SD of $d\delta$ (m)	3.25	6.24
Mean of $\|\mathbf{u}\|$	531.21	552.28
Mean number of steps	414	473
Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	2.197	2.611
(Success) Mean of $d\delta$ (m)	1.38	4.83
(Success) SD of $d\delta$ (m)	1.57	2.70
(Success) Mean of $\|\mathbf{u}\|$	531.16	554.60
(Success) Mean number of steps	421	414
(Success) Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	2.230	2.288

Table 6.17. Testing phase of a model trained normally on long distance waypoints for 2500 episodes.

Both of the models had a smaller success rate than the PID controller, especially the model 2500: it obtained 35.2% of success, which is very small. The model 1300 had 76% of success which is better, but still less than the results of the section 6.2: the SAC had managed to reach 86.4% of success on a long distance waypoints tracking task. This difference can come from the real-time setup of the simulation, which change the rate at which the SAC algorithm is sampling the environment and is sending its actions, resulting in a different AUV behaviour. **In this section the main focus will not be to beat the PID controller at all cost (unlike the previous section), but to experiment new ways of improving the learning of the task.** Here we can say that letting the training phase running for more episodes do not result in better performance: the model 2500 was trained on almost two times more episodes than the model 1300, but the success rate was then divided by more than two. Moreover, the collision rate and timeout failure rate are very high, compared to the previous testing phases, which confirms that the learning of the task needs to be improved.
We can see that the waypoints are further away than before thanks to the mean number of steps taken by the PID controller per episode: it was around 200 before, whereas now it is around 400. It is logic since the maximum distance between the waypoints and the AUV was doubled for this harder task.
Finally, only the model 1300 managed to save more energy than the PID controller by having smaller means of $\sum \|\mathbf{u}\|$.

Starting from this baseline trial where the SAC-based controller struggles to achieve as good

results as the PID controller, we now want to improve the learning of the task. These poorer results let room for improvement, and will allow to better observe the impact of the advanced training technique on the learning abilities of the SAC.

6.4.2 Learning from the PID controller

The first advanced learning technique we tried was *Learning from demonstration (LfD)*, taken from the subfield of Safe Reinforcement Learning and described in the section 5.2.1.2. These techniques are mainly used in order to have a safer training phase, but can also allows to converge faster towards a good behaviour. More specifically, these methods allow to derive a policy from a finite set of demonstrations provided by a teacher, before continuing the training in a normal way. The agent is then directly exposed to relevant regions of the state and action spaces, which allows to speed up the learning and to avoid dangerous areas of these spaces. Here the teacher will be a PID controller, establishing a link between the fields of control theory and machine learning.

6.4.2.1 Learning only on PID controller episodes

We first try to train the SAC algorithm only on demonstrations from a PID controller. The policy is not just initialize from the PID controller: it is entirely derived from it. During the training phase, the PID controls the RexROV 2 and fulfill the control task, which allows the replay buffer of the SAC to be filled exclusively with PID controller training samples. The SAC is trained simultaneously on these samples, but is not able to interact with the RexROV 2. The testing phase remains the same as before: the SAC is now controlling the AUV and is compared to the PID controller.

After 3000 training episodes, the PID controller had the following number of success during the training phase:

Number of success for each 100 episodes: [98, 95, 94, 94, 98, 91, 88, 94, 94, 95, 94, 94, 95, 91, 95, 95, 95, 91, 92, 93, 92, 93, 95, 97, 98, 92, 96, 92, 96, 100]

These success rates were produced only by the PID controller during the training phase, and are then as good as during the testing phase (since there is no difference between training and testing, from the point of view of the PID controller). These numbers do not provide any information about the convergence of the SAC algorithm trained in parallel.

The Figure 6.23 shows the cumulative rewards and the loss functions obtained during the training phase. At the contrary of the *Number of success for each 100 episodes*, this figure was produced by the training of the SAC algorithm. The total reward per episode corresponds

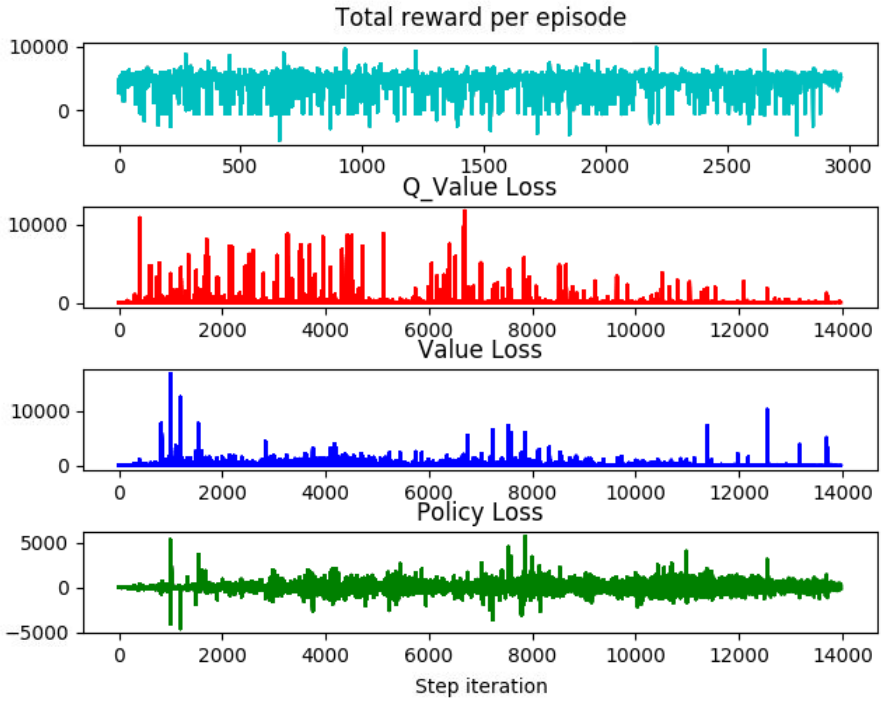


Figure 6.23. Total reward per episode and loss functions for a training only on PID controller episodes.

to the rewards obtained by the PID controlling the AUV, whereas the loss functions reflects the training of the SAC. These plots are very similar to the Figure 6.22, except that the total reward per episode is less often in the negative values of the graph.

Since the model 2500 completely failed to learn the task in the previous trial, we chose to test the models 1300 and 1750. Since we do not have information about how the learning process of the SAC went, we chose these models arbitrarily. Here are their respective results tables.

Metrics	PID	SAC
Success rate (%)	92.0	0.4
Collision rate (%)	7.2	48.8
Timeout failure rate (%)	0.8	50.8
Mean of $d\delta$ (m)	1.66	19.43
SD of $d\delta$ (m)	1.87	11.21
Mean of $\|\mathbf{u}\|$	528.56	539.57
Mean number of steps	395	749
Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	2.087	4.040
(Success) Mean of $d\delta$ (m)	1.36	4.13
(Success) SD of $d\delta$ (m)	1.46	1.56
(Success) Mean of $\|\mathbf{u}\|$	528.05	475.70
(Success) Mean number of steps	411	281
(Success) Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	2.164	1.332

Table 6.18. Testing phase of a model trained only on PID controller episodes, for 1300 episodes.

Metrics	PID	SAC
Success rate (%)	95.2	0.2
Collision rate (%)	4.2	91.8
Timeout failure rate (%)	0.6	8.0
Mean of $d\delta$ (m)	1.71	12.89
SD of $d\delta$ (m)	1.90	7.57
Mean of $\|\mathbf{u}\|$	529.48	536.00
Mean number of steps	417	548
Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	2.205	2.938
(Success) Mean of $d\delta$ (m)	1.42	5.54
(Success) SD of $d\delta$ (m)	1.52	2.72
(Success) Mean of $\|\mathbf{u}\|$	529.22	467.54
(Success) Mean number of steps	425	461
(Success) Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	2.246	2.151

Table 6.19. Testing phase of a model trained only on PID controller episodes, for 1750 episodes.

The SAC algorithms failed to learn the task from the PID training samples. The model 1300 had 2 success out of 500 episodes, and the model 1750 only had one.

Judging by the mean number of steps obtained during successful episodes, the only successes the SAC had were by reaching only close waypoints. Since the waypoints are placed randomly inside a given box, they can sometimes appears close to the AUV.

In the following trials, we will propose different implementations of the LfD in order to see if this method can result in better performances.

6.4.2.2 Bootstrapping the learning with the PID controller

After the previous failures, we try to *bootstrap* the training phase of the SAC with PID controller episodes. This means that the training phase will begin by a given number of PID training episodes, before switching to regular SAC training episodes. As before, the PID will first learn from training samples from the PID controller, before learning exclusively on its own training samples. When the learning process switch from PID episodes to SAC episodes, the replay buffer is emptied in order to not keeping the PID training samples, while the parameters of the neural networks are kept unchanged. This is how the LfD methods are usually implemented.
The threshold allowing to switch from PID to SAC is choose arbitrarily since no information are available about the training of the SAC during the PID episodes. We chose to switch after 600 PID training episodes.

After the first 600 PID training episodes, the SAC had the following success rates during 2600 training episodes:

Number of success for each 100 episodes: [5, 12, 21, 43, 38, 47, 52, 37, 7, 6, 0, 1, 8, 25, 24, 18, 35, 11, 17, 14, 35, 40, 4, 11, 7, 1]

Even if the agent started to explore the environment during the episodes carried out by the SAC algorithm, it did not managed to converge towards a good successful behaviour. **Initializing the parameters of the neural networks with PID examples did not improve the classical SAC training phase, and even harmed the learning process.**

The Figure 6.24 shows only the cumulative rewards and the loss functions obtained during the SAC training episodes. The total reward per episode plot is more often in the negative values than before. The Value Loss has only one peak on its plot, which is unusual. The learning process seems to be less active.

Since the SAC did not converge after the switch operated during the training phase (from PID episodes to SAC episodes), we did not lost time to test specific models from this trial. We used this time to test new configurations of the LfD method instead.

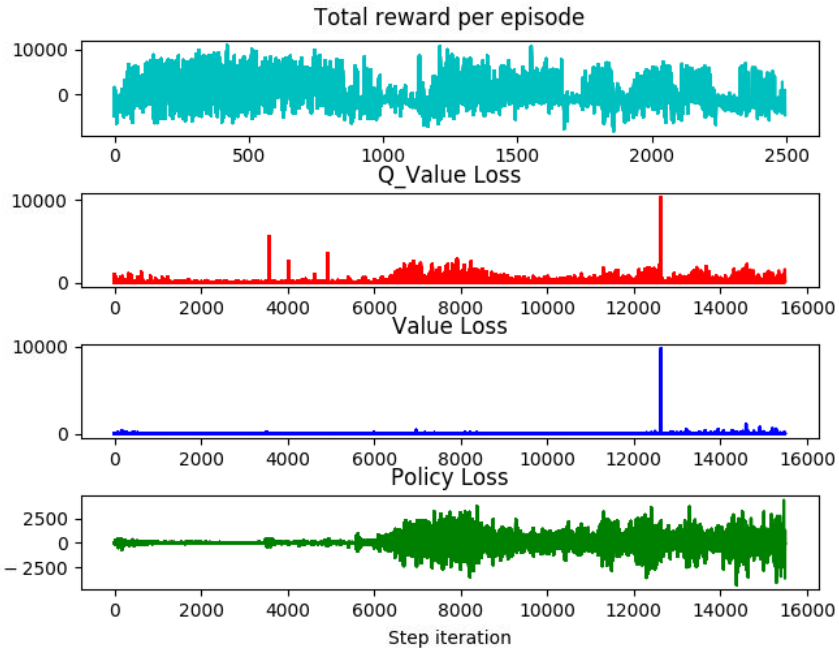


Figure 6.24. Total reward per episode and loss functions for a training bootstrapped by PID controller episodes.

6.4.2.3 Learning only on softer PID controller episodes

During the trials from the subsections 6.4.2.1 and 6.4.2.2, we noticed that the inputs generated by the PID controller during the training episodes were above the boundaries [-240: 240] of the SAC actions (defined in 6.1.3.2). The inputs u_i sent to the thrusters by the PID were around the boundaries [-300: 300]. This can biased the learning, since the SAC algorithm is learning inputs that sometimes cannot be replicated. This will lead the inputs computed by the SAC to be saturated, which would be an unexpected behaviour to the SAC.
We decided to modify the PID controller in order to make it output softer commands: here we call it the *softer PID controller*. The PID controller provided by the UUV Simulator has a parameter called the *max_thrust* in the thruster manager files, having a default value of 1540.0. We found experimentally that setting the max_thrust to 1000.0 would lead the output of the PID to be around the boundaries [-240: 240]. This modification is only valid for the learning process: max_thrust is set to 1000.0 during the training phase (in order to not biased the learning process), and is set back to 1540.0 during the testing phase (in order to compare the SAC with the same PID controller as in all the previous trials).

In this section, we decided to make the SAC algorithm learn only on softer PID controller episodes, similarly to the subsection 6.4.2.1. We want to see if the use of a softer PID controller can impact the results of the LfD approach.

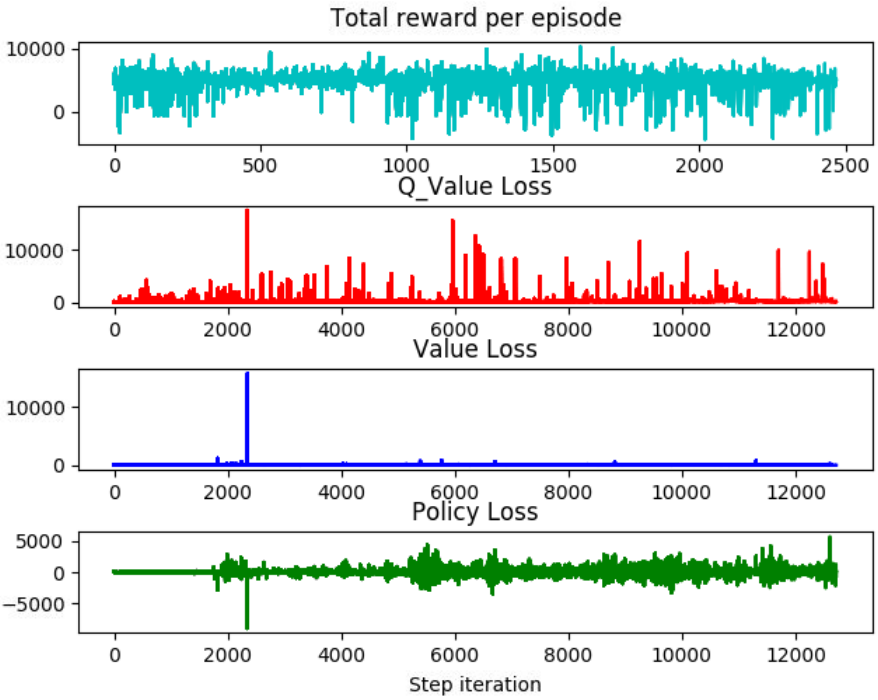


Figure 6.25. Total reward per episode and loss functions for a training bootstrapped by PID controller episodes.

After 2500 training episodes, the PID controller had the following number of success during the training phase:

Number of success for each 100 episodes: [87, 87, 90, 93, 100, 97, 99, 98, 96, 90, 91, 93, 93, 88, 88, 86, 83, 97, 84, 94, 95, 91, 92, 93, 91]

Like in the section 6.4.2.1, **these success rates are generated by the PID controller** and do not give any information on the learning process of the SAC algorithm.

On the Figure 6.25, we can see the total reward per episode generated by the PID, and the evolution of the three loss functions of the SAC. The plots are more similar to the Figure 6.24 than to the Figure 6.23, except that the peak of the learning is at the beginning of the training phase and not at the end.

We only chose to test the model 1300 in this trial, because of time limitations. Here is the corresponding results table.

Metrics	PID	SAC
Success rate (%)	94.8	0.2
Collision rate (%)	4.6	93.8
Timeout failure rate (%)	0.6	6
Mean of $d\delta$ (m)	2.64	17.93
SD of $d\delta$ (m)	3.48	11.42
Mean of $\|\mathbf{u}\|$	528.89	561.40
Mean number of steps	406	521
Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	2.150	2.925
(Success) Mean of $d\delta$ (m)	1.22	3.61
(Success) SD of $d\delta$ (m)	1.31	1.21
(Success) Mean of $\|\mathbf{u}\|$	528.74	516.74
(Success) Mean number of steps	412	326
(Success) Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	2.172	1.679

Table 6.20. Testing phase of a model trained only on PID controller episodes, for 1300 episodes.

The SAC did not have better results in this configuration since it obtained a success rate of 0.2% with a collision rate of 91.8%. This collision rate is the worst we get until now. Using a softer PID controller did not improve the performance provided by the LfD approach.

6.4.2.4 Bootstrapping the learning with a softer PID controller

In this subsection we are going to try to use the softer PID controller to bootstrap the training phase of the SAC. We kept the same *max_thrust* parameter set to 1000.0 for the training and 1540.0 for the testing. This time we chose to switch from PID only episodes to SAC episodes after 400 episodes.

After the first 400 PID training episodes, the SAC had the following success rates during 3000 training episodes:

Number of success for each 100 episodes: [4, 9, 20, 25, 28, 30, 30, 39, 38, 33, 46, 14, 1, 0, 34, 32, 47, 44, 13, 17, 17, 27, 30, 0, 11, 18, 21, 44, 34, 5]

Like in the section 6.4.2.2, **the SAC did not manage to converge towards a good behaviour**, since it did not even reach 50% of success rate.

The Figure 6.26 shows only the cumulative rewards and the loss functions obtained during the SAC training episodes. The plots are almost identical to the ones found in the Figure 6.24.

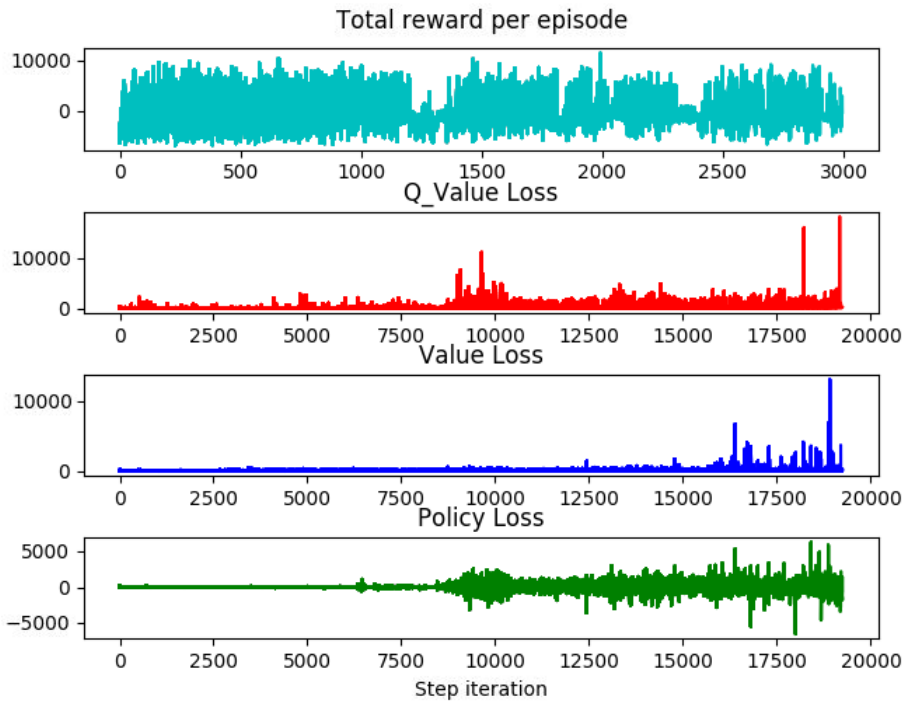


Figure 6.26. Total reward per episode and loss functions for a training bootstrapped by a softer PID controller.

Even if the training of the SAC algorithm did not go well, according to the results above, we tried to test two models in order to observe any changes in the performance. Here are the results tables of the models 1000 and 1750.

Metrics	PID	SAC
Success rate (%)	90.6	42.6
Collision rate (%)	8	16.6
Timeout failure rate (%)	1.4	40.8
Mean of $d\delta$ (m)	1.77	11.26
SD of $d\delta$ (m)	1.98	7.23
Mean of $\|\mathbf{u}\|$	529.40	414.21
Mean number of steps	406	729
Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	2.151	3.021
(Success) Mean of $d\delta$ (m)	1.43	5.90
(Success) SD of $d\delta$ (m)	1.56	3.23
(Success) Mean of $\|\mathbf{u}\|$	528.99	400.63
(Success) Mean number of steps	418	506
(Success) Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	2.205	2.024

Table 6.21. Testing phase of a model trained using the bootstrap of a softer PID controller, for 1000 episodes.

Metrics	PID	SAC
Success rate (%)	91.2	50.2
Collision rate (%)	8	29.6
Timeout failure rate (%)	0.8	20.2
Mean of $d\delta$ (m)	2.98	11.64
SD of $d\delta$ (m)	3.98	8.41
Mean of $\|\mathbf{u}\|$	528.62	553.28
Mean number of steps	391	592
Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	2.066	3.274
(Success) Mean of $d\delta$ (m)	1.23	5.35
(Success) SD of $d\delta$ (m)	1.30	3.10
(Success) Mean of $\|\mathbf{u}\|$	528.32	553.83
(Success) Mean number of steps	406	440
(Success) Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	2.139	2.431

Table 6.22. Testing phase of a model trained using the bootstrap of a softer PID controller, for 1750 episodes.

Learning from demonstration They reach a success rate of 42.6% and 50.2% respectively. This is a great improvement, since the others tested models had less than 1% of success rate.

However it remains clearly worst than the 76% of success rate obtained by the model 1300 in the section 6.4.1. We can also note that, at the contrary of the results found in the section 6.4.1, the model trained on the greater amount of episodes (1750) performed better here.
The use of LfD technics did not improve the initial performance of the SAC algorithm of the section 6.4.1.

6.4.3 Adding the batch normalization algorithm to the learning process

After trying the Learning from Demonstration approach, we try to use the *Batch Normalization* technique (described in detail in the section 3.1.2.3). The batch normalization method is an algorithm very used in the Deep Learning subfield, allowing to make the training of neural networks faster and more stable. The principle is to standardize the outputs of all the layers of the neural network with respect to each mini-batch.
It is not very often implemented in the neural networks composing the Deep Reinforcement Learning algorithms. The most known case is its implementation inside the neural networks of the Deep Deterministic Policy Gradient (DDPG, described in the section C.2.2). We want to see if batch normalization can affect the learning process of the SAC algorithm.

We simply applied the batch normalization method to the soft Q-value network, the soft value network and the policy network. We used the default implementation provided by *Pytorch* for each layer of these neural networks. Its implementation is the same as explain in the section 3.1.2.3, without the optional fifth step. We have the following hyperparameters values: $\epsilon = 10^{-5}$, $\gamma = 1$ and $\beta = 0$. The remainder of the training and the testing phases is the same as in the section 6.4.1.

After 5000 training episodes, the SAC had the following number of successes during the training phase:

Number of success for each 100 episodes: [4, 6, 10, 18, 34, 41, 56, 62, 67, 61, 65, 73, 72, 71, 72, 82, 78, 77, 79, 74, 79, 77, 86, 81, 90, 76, 86, 77, 76, 70, 77, 81, 79, 80, 82, 80, 81, 93, 86, 77, 80, 80, 83, 75, 84, 79, 81, 80, 84, 83]

We let the training phase continue until the maximum number of training episodes (5000) in order to see if the Batch Normalization affect the stability of the learning phase. We can see that the value of the success rate never decreased drastically during the whole training phase. In the previous trials, we could often observe several success rate drops in the middle of the learning process. **Here the learning process has been stabilize thanks to the use of the Batch Normalization algorithm.**

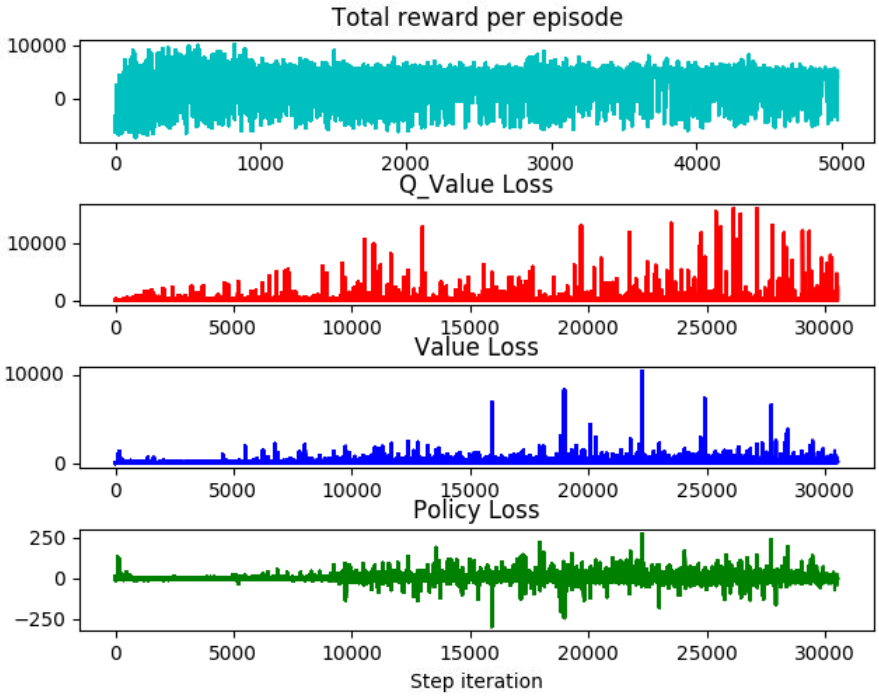


Figure 6.27. Total reward per episode and loss functions for a training using batch normalization.

The training plots of the Figure 6.27 are classical. The total reward per episode is especially noisy.

Since we let the training process go very far for the first time, we tested a model trained on 3750 episodes. We want to see if such a large number of training episodes can lead to a good behaviour. We also tested the model 1300. Here are the results tables.

Metrics	PID	SAC
Success rate (%)	94.1	72.9
Collision rate (%)	4.7	6.7
Timeout failure rate (%)	1.2	20.4
Mean of $d\delta$ (m)	4.48	6.46
SD of $d\delta$ (m)	6.74	4.25
Mean of $\|\mathbf{u}\|$	528.21	347.51
Mean number of steps	401	570
Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	2.119	1.981
(Success) Mean of $d\delta$ (m)	1.21	4.23
(Success) SD of $d\delta$ (m)	1.35	2.14
(Success) Mean of $\|\mathbf{u}\|$	527.65	349.97
(Success) Mean number of steps	406	480
(Success) Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	2.140	1.676

Table 6.23. Testing phase of a model trained using batch normalization, for 1300 episodes.

Metrics	PID	SAC
Success rate (%)	92.6	83.2
Collision rate (%)	5.2	5.8
Timeout failure rate (%)	2.2	11
Mean of $d\delta$ (m)	4.48	6.04
SD of $d\delta$ (m)	6.27	4.14
Mean of $\|\mathbf{u}\|$	530.18	370.59
Mean number of steps	412	483
Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	2.186	1.791
(Success) Mean of $d\delta$ (m)	1.42	4.47
(Success) SD of $d\delta$ (m)	1.52	2.57
(Success) Mean of $\|\mathbf{u}\|$	529.19	376.38
(Success) Mean number of steps	413	431
(Success) Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	2.179	1.620

Table 6.24. Testing phase of a model trained using batch normalization, for 3750 episodes.

The model 3750 obtained better results than the model 1300 in all aspects of the performance: it had better success, collision and timeout failure rates, while having better mean

and standard deviation for the distance error $d\delta$, as well as a better mean of $\sum \|\mathbf{u}\|$ ($.10^5$). Unlike the results of the section 6.3, letting the SAC train on more episodes led to better results. **It confirms that the batch normalization algorithm allowed to stabilize the training process: the greater number of training episodes did not result in a drop in performance during the testing phase.**
The model 3750 really succeeded to learn the task and the dynamics of the RexROV 2, compared to the previous models. **It achieved a success rate of 83.2% which is the best one of the section 6.4 so far.** It did not beat the PID controller on the success rate metric, but **it managed to save more energy according to the mean of $\sum \|\mathbf{u}\|$ metric.** Moreover the SAC had here its least number of collisions (5.8%) on this harder task, which means that the testing phase is safer than before.
The success rate of this model is slightly lower than the one of the initial trial (86.4%) from the section 6.2, however the PID controller did also worse in this trial (92.6%) than during the initial trial (96%). The comparison is then biased, since these controllers were not exposed to the exact same test episodes. If we compute the difference between the success rate of the PID and the success rate of the SAC in both trials, we have a difference of 9.4% for this trial and a difference of 9.6% for the initial trial. **It means that the SAC algorithm is almost as close to the PID controller as it was in the initial trial.**

Unlike the Learning from Demonstration approach, **the batch normalization algorithm allowed to greatly improve the performance of the SAC algorithm on this task.** These improvements come from its ability to stabilize and speed up the learning process of neural networks, like in Deep Learning tasks. The original paper of the DDPG [212] also explains that **the use of batch normalization in robotics allows to normalize the different physical units found inside the state vector and to be robust to the parameters variations which can appear across multiple robotic platforms.**

6.4.4 Using the Batch Normalization with the Learning from Demonstration approach

We previously tested two advanced training techniques in order to improve the learning process of the SAC algorithm, which did not give the same results. The Learning from Demonstration approach used with the PID controller as a teacher failed completely, and even made the performance of the SAC much worst. Conversely, the Batch Normalization really improved the learning of the task. In this subsection, we want to try to implement both of them at the time during the training phase of the SAC algorithm.

6.4.4.1 Learning only on PID controller episodes, with the batch normalization

For this trial, we simply implemented the same Learning from Demonstration approach from the section 6.4.2.1 with the default PID controller: the SAC algorithm is learning only on PID controller episodes during the entire training phase. We also add the batch normalization method to the layers of the three neural networks of the SAC algorithm, like in the section 6.4.3.

After 5000 training episodes, the PID controller had the following number of success during the training phase:

Number of success for each 100 episodes: [91, 91, 95, 84, 96, 93, 91, 93, 92, 94, 89, 90, 91, 88, 94, 92, 93, 91, 93, 95, 91, 92, 94, 89, 93, 85, 91, 93, 93, 92, 89, 91, 92, 88, 92, 94, 88, 95, 96, 97, 91, 93, 92, 85, 93, 93, 92, 95, 97, 92]

Like previously, **these success rates are generated by the PID controller** and do not give any information on the learning process of the SAC algorithm.
In the Figure 6.28, the total reward per episode plot were generated by the PID, whereas the loss functions corresponds to the SAC. These plots are similar to the previous ones, except the policy loss which shows a large peak in the negative values. All the non-zero values of these loss functions means that the learnng process is working.

We tested three models: the models 600, 1300 and 3750. Here are the results tables of the models 600 and 1300.

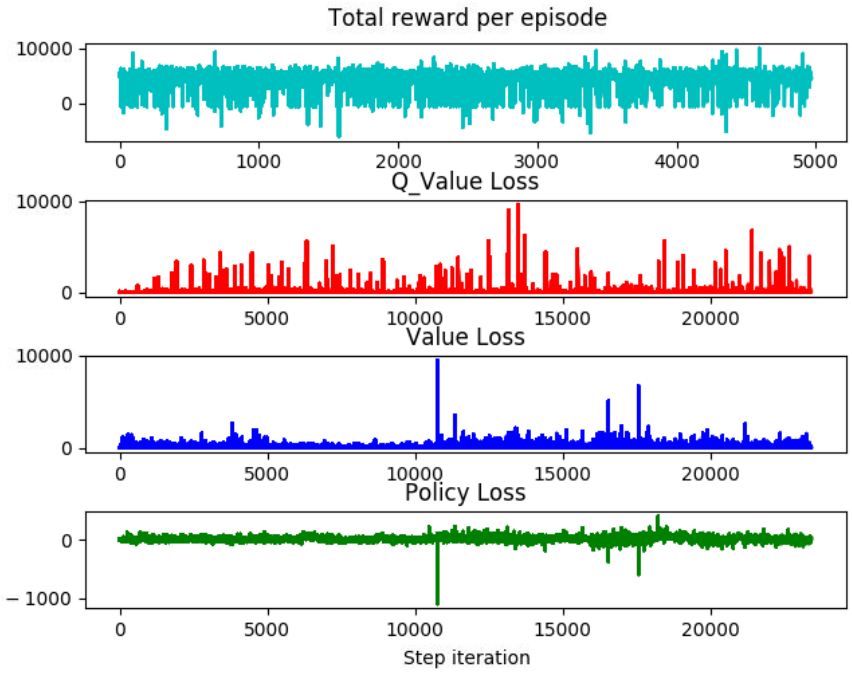


Figure 6.28. Total reward per episode and loss functions for a training using batch normalization with only PID episodes.

Metrics	PID	SAC
Success rate (%)	96.4	0.4
Collision rate (%)	2.2	49.2
Timeout failure rate (%)	1.4	50.4
Mean of $d\delta$ (m)	1.75	16.43
SD of $d\delta$ (m)	1.98	9.12
Mean of $\|\mathbf{u}\|$	528.85	395.03
Mean number of steps	410	780
Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	2.169	3.083
(Success) Mean of $d\delta$ (m)	1.41	7.90
(Success) SD of $d\delta$ (m)	1.54	3.48
(Success) Mean of $\|\mathbf{u}\|$	528.47	340.28
(Success) Mean number of steps	408	555
(Success) Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	2.149	1.883

Table 6.25. Testing phase of a model trained only on PID controller episodes and using batch normalization, for 600 episodes.

Metrics	PID	SAC
Success rate (%)	91.0	0.0
Collision rate (%)	6.6	37.4
Timeout failure rate (%)	2.4	62.6
Mean of $d\delta$ (m)	3.96	19.15
SD of $d\delta$ (m)	5.63	11.35
Mean of $\|\mathbf{u}\|$	528.75	427.58
Mean number of steps	411	844
Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	2.175	3.611
(Success) Mean of $d\delta$ (m)	1.12	X
(Success) SD of $d\delta$ (m)	1.24	X
(Success) Mean of $\|\mathbf{u}\|$	527.58	X
(Success) Mean number of steps	415	X
(Success) Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	2.185	X

Table 6.26. Testing phase of a model trained only on PID controller episodes and using batch normalization, for 1300 episodes.

We did not show the results table of the model 3750, because we noticed during the testing phase that the SAC algorithm only had 1 success out of 225 episodes. **We chose to stop the testing phase before its end, in order to save time for running other trials.** The results of this model were not worth the computing time of the simulations.

The models 600 and 1300 had very poor success rates (0.4% and 0% respectively), we are not going to described further these results. Since the model 1300 had 0% of success rate, the *(Success)* version of the metrics have no values in the *SAC* column of the results table.
Despite the good results provided by the batch normalization algorithm implemented alone, this technique is not sufficient to compensate the poor results of the Learning from Demonstration approach used with the PID controller.

6.4.4.2 Learning only on softer PID controller episodes, with the batch normalization

In this trial, we did the same implementations as during the previous section but with a *softer* PID controller: its *max_thrust* parameter is set to 1000.0 instead of 1540.0. Here we combined the approaches from the sections 6.4.2.3 and 6.4.3.

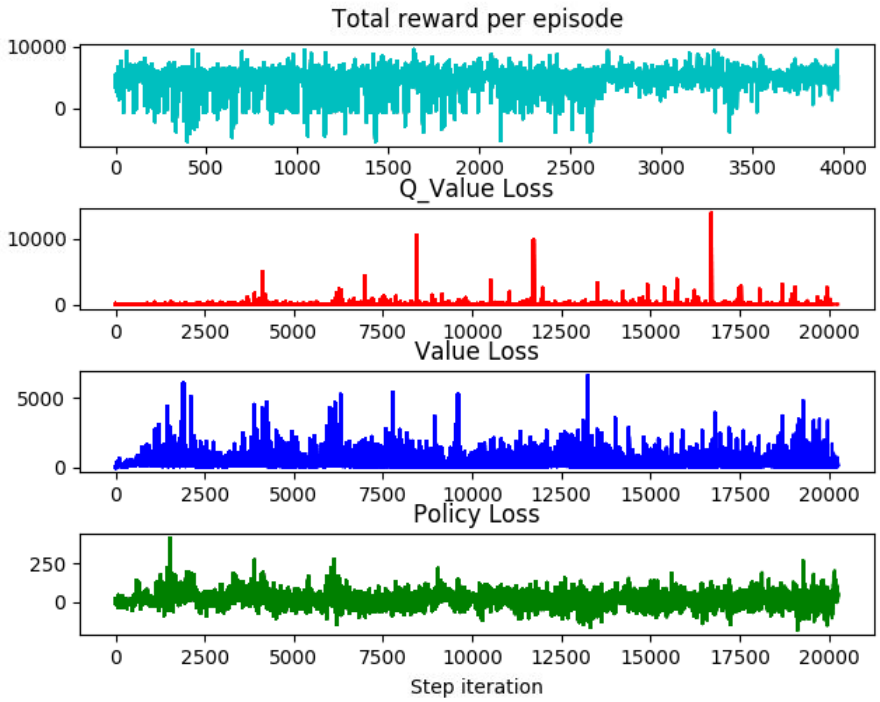


Figure 6.29. Total reward per episode and loss functions for a training using batch normalization with only softer PID episodes.

After 4000 training episodes, the PID controller had the following number of success during the training phase:

Number of success for each 100 episodes: [94, 89, 88, 88, 83, 91, 91, 85, 90, 87, 91, 85, 81, 88, 85, 85, 92, 91, 87, 93, 94, 86, 90, 91, 92, 86, 92, 98, 96, 100, 97, 96, 97, 87, 98, 97, 96, 96, 98, 98]

Like usually, **the PID controller generated the success rates above** and the total reward per episode plotted on the Figure 6.29, while the SAC algorithm generated the three other plots of its loss functions.

We chose to test the models 600 and 1300. Here are their results table.

Metrics	PID	SAC
Success rate (%)	97.4	0.2
Collision rate (%)	1.4	7
Timeout failure rate (%)	1.2	92.8
Mean of $d\delta$ (m)	1.39	18.99
SD of $d\delta$ (m)	1.60	11.37
Mean of $\|\mathbf{u}\|$	527.97	397.43
Mean number of steps	415	967
Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	2.189	3.841
(Success) Mean of $d\delta$ (m)	1.16	3.39
(Success) SD of $d\delta$ (m)	1.29	1.36
(Success) Mean of $\|\mathbf{u}\|$	527.66	423.97
(Success) Mean number of steps	414	662
(Success) Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	2.177	2.802

Table 6.27. Testing phase of a model trained only on a softer PID controller episodes and using batch normalization, for 600 episodes.

Metrics	PID	SAC
Success rate (%)	96.0	0.0
Collision rate (%)	3.0	47.0
Timeout failure rate (%)	1.0	53.0
Mean of $d\delta$ (m)	2.62	20.04
SD of $d\delta$ (m)	3.35	12.28
Mean of $\|\mathbf{u}\|$	529.34	413.50
Mean number of steps	427	770
Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	2.260	3.185
(Success) Mean of $d\delta$ (m)	1.37	X
(Success) SD of $d\delta$ (m)	1.48	X
(Success) Mean of $\|\mathbf{u}\|$	528.86	X
(Success) Mean number of steps	431	X
(Success) Mean of $\sum \|\mathbf{u}\|$ $(.10^5)$	2.273	X

Table 6.28. Testing phase of a model trained only on a softer PID controller episodes and using batch normalization, for 1300 episodes.

We stopped the testing phase before the 1000 episodes (500 for each controller), since we saw

that both of these models had poor results again. The metrics of the model 600 were computed on 419 episodes for each controller, whereas the metrics of the model 1300 were computed on 406 episodes for each controller (instead of 500). Moreover, since the model 1300 had 0% of success rate, the *(Success)* version of the metrics have no values in the *SAC* column of the results table.

Like in the previous subsection, **both trained models obtained very poor results. The use of a softer version of the PID controller did not manage to make the Batch Normalization and Learning from Demonstration techniques work together.**

6.4.5 Sum up of the results on the harder task

The Batch Normalization algorithm was the only advanced learning technique which allowed to improve the performance of the SAC algorithm on the long distance waypoint tracking task (the harder task). With the model 3750, it reached a success rate of 83.2% while managing to save a lot of energy by having one of the lowest mean of $\sum \|\mathbf{u}\|$ ($1.620 * 10^5$) of the section 6.4, if we only take into account the successful episodes. **Batch normalization managed to stabilize the learning process (training on more episodes did not result in drop in performance), while allowing to be robust to the parameters variations which can appear across multiple robotic platforms.** This is a really useful technique for controlling a robot with deep reinforcement learning algorithms.

The Learning from Demonstration (LfD) approach did not work at all. The SAC is an entropy-regularized deep RL algorithm, meaning that its main equations have been modified in order to take an entropy term into account (see section 3.2.3.3). The LfD methods described in the section 5.2.1.2 have all been applied to non-entropy-regularized algorithms. We came to the the conclusion that the initialization methods like the LfD approaches cannot be straightforward applied to entropy-regularized algorithms.
Indeed, the entropy term is used in order to encourage the exploration of the environment, while maximizing the cumulative rewards at the same time. When the neural networks of the SAC are initialized with PID controller examples, these examples shows only the best way to go towards the target waypoints. These examples display how to get the large cumulative rewards but not how to better explore the environment. In conclusion, **the SAC expects to get training examples displaying these two objectives, so the LfD methods need to be modified in order to take the exploration of the environment into account.**
After investigating the literature, we found [155]: a very recent work (currently in preprint) proposing an entropy-aware model initialization method for deep RL. This can be a good start in order to understand how to modified the initialization methods in order to apply them to entropy-regularized algorithms.

Even if the best models of this section obtained lower success rates than the models found in

the section 6.2, **these results were obtained with a real-time simulation. This allows to facilitate the transfer from simulation to real-world.**

6.5 A methodology for the end-to-end control of AUVs based on deep RL

Thanks to all the trials performed in this chapter, we can now propose a methodology for the training of deep reinforcement learning algorithms in order to perform AUV waypoint tracking tasks. After choosing the deep RL algorithm we want to apply to this control task, we can follow the following steps:

1. During the first trials, the chosen DRL algorithm should be implemented using the default configuration proposed by its corresponding reference paper. After several trials, we can start to adapt either the hyperparameters of the DRL-based controller, or elements of the environment, in order to improve the learning process.

2. After determining the main hyperparameters and environment settings, we can add the Batch Normalization algorithm to the implementation. Batch Normalization will always provide better results and cannot harm the learning process.

3. When the number of successful episodes starts increasing during the training phase, we can test some models. We can successively test models taken from different times of the training phase, in order to cover a broader range of trained models.
   The trained models should be selected using the success rates obtained during the training phase (defined in the section 6.1.3.3). The higher the success rate is during the training phase, the better the model will probably be during the testing phase. Beyond a training sucess rate of 70%, we can start to expect great results from the model during the testing phase. Below a training sucess rate of 40%, the test of the model may not be worth, and can constitute a waste of time and computational resources.

4. If the success rates obtained during the testing phases do not match our expectations, the control task can be decomposed in multiple simpler tasks. For example if the DRL-based controller is not able to make the AUV reach long range waypoints, we can try to make it reach several intermediate waypoints instead. If the SAC is not able to reach waypoints located at 50 meters, it can instead try to reach two successive waypoints located at 25 meters. If it still does not work, we can further decomposed the task, by trying to reach two waypoints located at 20 meters, then one waypoint located at 10 meters, and so on.

5. Once we managed to obtain satisfactory success rates during the testing phases, we can try to improve other performance metrics such as the power consumption, the tracking

abilities or the time needed to fulfill the test episodes. Moreover we can also try to reduce the size of the state vector in order to eventually remove sensors from the robot and to reduce the production costs.

The previous methodology can be used in order to perform an end-to-end control of the AUV: the deep RL algorithm is simultaneously performing the low-level control and the high-level control (or guidance) of the AUV. During our trials, the SAC-based controller was performing end-to-end control since no guidance algorithms were used: the SAC were generating the trajectories which must be followed by the AUV.
This end-to-end control approach worked well on the simpler task (section 6.3), but failed to match the performance of the PID controller during the harder task (section 6.4). When the deep RL algorithm fails to match the PID controller, we can add intermediary waypoints to the task: instead of reaching one target waypoint, the AUV has to successively reach closer waypoints. The original path is divided into several smaller trajectories. On each of these small trajectories, the deep RL algorithm can perform an end-to-end control. An additional objective of the training procedure can be to determine the maximum range until which the deep RL algorithm can perform an end-to-end control of the AUV successfully.

Chapter 7

Conclusions

The first part of this concluding chapter aims to further analyze the results obtained during this work, and to make the link between our proposals and the existing state of the art. It answers the questions raised in the introduction about the objectives we defined. The second part describes the main ideas about what we expect to do in the future in order to further improve this work.

7.1 Summary of the objectives fulfilled by our proposals

In this section, we are going to summarize our results and to compare them with the introduction chapter and the state of the art. Our observations have been grouped into several thematic sections, in order to segment our contributions based on the objectives they aim to fulfill.

Developing a simulation architecture for the application of RL to marine robotics

At the beginning of this work, we did not have a simulation tool responding to all our needs. We had to connect together existing components and to adapt them to our problem. Now we have a customizable simulation platform based on the tools described in the section 6.1.1. This platform can be reusable for a variety of tasks involving marine robotics and/or reinforcement learning.

Making the SAC converge to a correct behaviour

First of all our main objective was to analyze if the SAC algorithm is able to understand the dynamics of an AUV and to learn how to perform a waypoint tracking task, while dealing with the varying ocean currents and the noises found in the sensors and the actuators

of the RexROV 2.
In all of our tests, we managed to make the SAC successfully converge towards a correct behaviour a great number of times, especially in the section 6.3. During the testing phases of this simplified task (where the waypoint is at a distance of at most 20 meters from the AUV on the X and Y axes, instead of 50 meters), it reached a success rate of more of 90% for a lot of trials.
Moreover, the SAC is able to understand the dynamics induced by the particular layout of the AUV thrusters (described in the section 6.1.2.1), without the need of a thruster manager and a thruster allocation matrix like the implementation of the PID controller found in the UUV Simulator (see the section 6.1.3.1). This demonstrates the effectiveness of the learning abilities of the SAC algorithm.
We can also note that the SAC algorithm is a model-free algorithm, which means that the Fossen's models (described in the section 4.1.1) are not needed during the design of this AUV controller. This is a great feature, since it allows this controller to be applied to other type of dynamical systems (not only to AUVs). It will only required to tune some of the hyperparameters.

Before our first publication [309], the SAC had never been tested for the control of AUV: our first objective has been achieved, allowing to state that an AUV can be controller by a SAC-based controller. We managed to found a reward function that is able to match the needs of the control task and the AUV.
Moreover the waypoint tracking task is a very general task. Since this SAC-based controller is able to reach target waypoints, it can be applied to any control tasks involving waypoint tracking or path-following. Indeed, any trajectory can be decomposed in several waypoints.

Outperforming the PID controller

During all of our trials, the SAC-based controller has been compared with PID controllers. Several performance metrics were used in these testing phases in order to measure various aspects of the performance: the percentage of success computed from the number of reached waypoints, the power consumption, the deviation from the expected paths, etc.
In many tests presented in section 6.3, the SAC algorithm managed to equalize and even sometimes to surpass the PID controller in terms of success rate. Moreover, even if the SAC-based controller took systematically more time to reach the waypoints than the PID, it almost always consume less energy. This was due to the fact that the SAC generated average thrusters input signals of smaller magnitudes. For a given test episode, the sum of all the inputs sent by the SAC to the AUV remained smaller than the PID controller's inputs.
We can also note that generally the SAC algorithm deviates more from the expected path (what we named the *ideal trajectory*) than the PID controller. These deviations (measured in our results by the *distance error* $d\delta$ metric) are due to the sudden variations found in both the magnitudes and the directions of the ocean currents. The SAC was able to adapt its behaviour

to these unexpected external disturbances and to still reach the waypoints despite these large deviations. Moreover, the SAC-based controller was also able to control the AUV despite the noises added to the sensors and actuators variables. All these disturbances have been detailed in section 6.1.2.2.

Even if the target waypoints can take more time steps to be reached, the SAC managed to outperform the PID controller in terms of power consumption, while still having at least as much of success rate as it. It was able to find a trade-off between performance and consumption.

Reducing the size of the state vector

One of the main goals of the section 6.3 was to analyze how the composition of the state vector $\mathbf{S}_t$ given to the SAC agent can affect the performance of the deep reinforcement learning-based controller. A detailed sum up if these trials can be found in the section 6.3.3.
During the simplified task where the target waypoints are closer to the initial position of the AUV, we began our trials with an initial 23-dimensional state vector. This vector included typical AUV sensor measurements (measuring the variables described in the section 6.1.2.1), as well as information about the tracking errors with respect to the target waypoints (usually given by a guidance algorithm).

We managed to progressively reduce the size of the state vector by removing variables trials after trials, and the SAC-based controller could still fulfill the waypoint tracking task in most cases. The lowest configuration was reached by setting the state vector to 10 variables (section 6.3.2.6):

$$\mathbf{S}_t = [\ \psi_e,\ \boldsymbol{x}_e,\ \boldsymbol{u}_{t-1}\ ]^T \tag{7.1}$$

where ψ_e is the tracking error of the yaw angles, $\boldsymbol{x}_e$ is the error vector between the position vector $\boldsymbol{x}$ and the position vector of the waypoint, and $\boldsymbol{u}_{t-1}$ is the vector of the past inputs (the inputs sent by the controller at the previous time step). In this configuration, the controller was able to obtained a success rate of 97.6%, which is almost a perfect score.
As stated in section 6.1.2.2, we focused only on low-level control, meaning that we assume that the variables found in the state vector are provided by guidance and navigation algorithms from other works. We designed a deep reinforcement learning-based controller, which is one of the component of the GNC system of the AUV (see section 2.1.2.2).

We can compare our results with some of the deep reinforcement learning-based controllers presented in section 4.2.2.
For example, the authors of [51] used the DDPG algorithm in order to control the low-level

variables of an AUV, such as the linear and angular velocities. They used the state vector detailed in the equation 4.7, which was composed of 23 variables. The state vector of [51] has the same size as our initial state vector. Both of the tasks deal with the low-level control of an AUV equiped with six thrusters. The only difference is that the authors of [51] are applying velocities regulation to their AUV, whereas we are applying position regulation to the RexROV 2.
In [125], the authors applied the PPO algorithm to an AUV in order to perform high-level control (or guidance) tasks such as path-following and collision avoidance. The state vector given to the agent was composed of 14 dimensions, which is bigger than our smallest 10-dimensional vector. Moreover, the PPO implemented in [125] needed the velocity vector of the surrounding ocean currents in order to generate correct trajectories, whereas our SAC-based controller was able to reach the target waypoints without any information about varying ocean currents.
The guidance algorithms generally need fewer information than the controllers, since they do not need control data such as the past inputs or the tracking errors. They have to take high-level decisions and to generate reference signals for the low-level controllers. The low-level controllers need more variables than the guidance component, in order to control the dynamics of the AUV and to follow these reference values.
Being able to perform a waypoint tracking task with a state vector composed of only 10 variables is a great improvement, since we managed to perform a low-level control task with fewer variables than a lot of guidance algorithms.

This study also allowed to state that the presence of the yaw error ψ_e and the past inputs $\boldsymbol{u}_{t-1}$ inside the state vector is really important for the success of the learning process. When the yaw error ψ_e was removed from the state vector (in section 6.3.2.7), the learning process completely failed and the SAC algorithm was not able to converge towards a satisfactory behaviour. When the vector of the past inputs $\boldsymbol{u}_{t-1}$ was removed from the state vector (in the section 6.3.2.8), the SAC converge towards a sub-optimal behaviour and the success rate obtained during the testing phase dropped to 75.4%.
In our last trial (presented in the section 6.3.2.9), we also saw that the SAC cannot learn to reach the waypoints with only the pitch error θ_e and the yaw error ψ_e. It will always need to know the position tracking errors $\boldsymbol{x}_e$.

Finally we can also add that with a reduced state vector, less sensor measurements are needed. This means that we can equip the AUV with fewer sensors, which will reduce the production costs.

Experimenting advanced techniques for the learning process

In section 6.4, we experimented advanced learning techniques in order to improve the training and the performance of the SAC algorithm in a harder task than before. The target

waypoints were placed to a distance of 50 meters from the AUV (on the X and Y axes), instead of 20 meters as in the section 6.3. Two main approaches were tested: the use of the Batch Normalization (BN) algorithm and the Learning from Demonstration (LfD) method.

BN is popular among the deep learning community, since it allows to avoid the gradients to converge towards extreme values during the backpropagation of the neural networks (see section 3.1.2.3). BN is rarely used in deep RL, and we could only found it in the implementation of the DDPG algorithm (see section C.2.2).
In section 6.4.3, the use of BN allowed to improve the performance of the SAC-based controller on this harder task and to stabilize the learning process: no more drops were found in the success rates obtained during the training phases, as it was the case before.
Moreover the application of the BN algorithm to be robust to the variations which can appear in some parameters when the controller is applied to different robots.
It seems that BN can never harm the learning process and that it should be always implemented inside deep reinforcement learning algorithms.

LfD is an approach allowing to derive a policy from a set of demonstrations. These demonstrations are produced by a teacher, which can be a human, as well as another algorithm. We tried to apply LfD to the learning process of the SAC algorithm, by setting the PID controller as the teacher.
During the trials of the sections 6.4.2 and 6.4.4, the SAC algorithm was trained with episodes generated by a PID controller. We tried multiple methods in order to LfD to this task:

- Training the SAC only on PID episodes during the entire training phase.
- Bootstrapping the training phase with PID episodes, before continuing the learning process with normal SAC episodes
- Coupling the LfD approach with the BN algorithm
- Using a PID controller generating lower inputs (called the *softer PID controller*).

Despite all our efforts, we could not make the SAC algorithm converge towards a satisfactory behaviour using the LfD approach. The best case was achieved when we bootstrapped the learning process with 400 episodes of a softer PID controller (in the section 6.4.2.4), where one of the model obtained a success rate of 50.2%.
We concluded that this approach can only skew the learning process of the SAC algorithm. As explained in section 3.2.3.3, SAC is an entropy-regularized algorithm. This means that a new term (based on an entropy measure) was added to the traditional reinforcement learning objective function in order to encourage the exploration of the unknown environment. The SAC algorithm has thus to simultaneously maximize the cumulative rewards (by fulfilling the task) and the entropy (by exploring its environment). The problem is that the episodes

produced by the PID controller only show how to fulfill the task (by reaching directly the target waypoints). Consequently, when the SAC algorithm wants to maximize the entropy by exploring the environment, the PID controller never offers this opportunity. The PID controller only wants to fulfill the task and has no notion of entropy regularization. This causes the failure of the learning process.
The LfD approach cannot be applied to entropy-regularized reinforcement learning algorithms the same way as the others RL algorithms. As explained in section 5.2.1.2, LfD approach has only been applied to non-entropy-regularized reinforcement learning algorithms and its application to SAC is not as straightforward as in the literature.

A methodology for the training of deep reinforcement learning algorithms for the end-to-end control of AUVs

In section 6.5, we proposed a methodology allowing to perform the end-to-end control of AUV: our SAC-based controller was able to fulfill the features of both the low-level control and high-level control of the AUV. If this end-to-end control is too difficult to maintain for long range waypoints, the task can be decomposed in multiple simpler tasks by inserting closer waypoints along the original path.

Being able to carry out an end-to-end control of the AUV is an advantage over the PID controller, since the PID is only performing a low-level control. In our testing phase, we paired the PID with a simple guidance method consisting in straight lines going from the initial position of the AUV to the target waypoints. This guidance approach does not allow the controller to be flexible and the AUV cannot adapt to the marine environment.

This methodology could be generalized to other types of robots (aerial, terrestrial) and to other type of control task. All control tasks involving to produce a movement from a given system could be suitable to the procedures we defined.

Facilitating the transfer towards real-world robots

Since the beginning of this work, efforts have been made in order to facilitate the transfer of our results from our simulation tools to real-world robotic platforms.

First of all, we chose the ROS middleware (described in section 6.1.1) which is known for providing a great flexibility and for allowing to easily transfer its software components (the *nodes*) towards embedded systems. Its component-based architecture also allows to easily change the type of robot, sensors and of actuators we are using, as well as the guidance and

the navigation algorithms of the GNC system (see section 2.1.2.2).

Moreover as explained in section 6.3.1, we set the simulation to be computed in real-time. After many observations, we noticed that changing the speed at which the simulation is computed can change the behaviour of the SAC algorithm. This is due to the fact that ROS is executing its nodes in an asynchronous way: when the simulation is computed at a different speed, the SAC (executed in an independant node) is sampling the environment at a different rate than before the change, and its actions are not maintained during the same number of time steps.
This means that a DRL-based controller which obtained good results during the simulated testing phases will not necessarily perform the same when it will be embedded on a real robotic platform. From the point of view of the agent of the DRL algorithm, the environment is not acting the same as during its training phase.
Executing the simulation in real-time is guaranteeing that the behaviour of the SAC-based controller will be the same on a real-world robot as during our simulation results.

Finally, the three neural networks (NN) implemented by the SAC algorithm (detailed in section 6.1.3.2) are shallow networks: they are all composed of only two hidden layers of 256 neurons. These are very light NN in terms of the amount of parameters needed to be updated during the learning process.
Moreover the smaller the state vector is (like during the results of section 6.3), the smaller the input layer of these NN will be. This means that less memory usage and computational resources will be needed.
These two elements allow the SAC algorithm to be embedded inside platforms with fewer memory (RAM) and computational (CPU) resources. This DRL-based controller can thus be applied to a broader range of robots. In order to give an idea to the reader of the memory usage taken by this DRL-based controller, the SAC algorithm took 3,6 GB of RAM and 600 MB of V-RAM (the RAM dedicated to the GPU).

We also recall that the Batch Normalization technique we implemented inside the SAC algorithm allows to be robust to the changes appearing for some parameters, when the controller is applied to different robots.

7.2 Future works and openings

We managed to achieve good results and to propose a general training methodology for the end-to-end control of robots. Here is a list of the short-term and medium-term objectives we would like to achieve based on this work:

- To implement more realistic external disturbances inside the simulations. During this

work, the angles and the magnitudes of the ocean currents were randomly sampled : these parameters were updated every 100 time steps using a uniform probability distribution across their whole ranges of value. This does not reflect the real-world marine environment, and realistic hydrodynamics models should be try for generating the ocean currents of the simulation. Moreover, the noise values added to the sensors and actuators signals were also randomly sampled every 100 time steps, and more characteristic models of noise should be used. These models could be found by investigating more deeply the hydrodynamics literature.
Another interesting approach could be tried by using unrealistic disturbances during the training phase, and realistic disturbances during the testing phase. The SAC algorithm could be trained using randomly sampled ocean currents and noise signals (like we did during this work), before being compared to the PID controller using realistic hydrodynamics models. This method could allow to evaluate if the SAC is able to generalize the learnt behaviour from unrealistic simulations to realistic simulations. This could constitute an intermediary step before deploying the controller on a real-world robot. If this method would give poor results during the testing phase, realistic hydrodynamics models should be implemented during both the training phase and the testing phase (in order to facilitate the transfer from simulation to real-world).

- To implement the SAC-based controller we developed inside a real-world AUV. Thanks to our efforts for facilitating the transfer towards the real world, we will only have to change several hyperparameters in order to make the transfer from simulation to real world, even if the real AUV would not be the same as the RexROV 2.

- To test more advanced training techniques in order to compare their impact on the learning process with BN and LfD methods. One of the first approach we would like to experiment is implementing new types of experience replay mechanisms [82][376], such as *Prioritized Experience Replay (PER)* for example. The principle of PER [292] is to give a greater probability of being sampled from the replay memory buffer to the transitions having the largest TD errors (these definitions are explained in section 3.2.2.3). The sample that got a large TD error, means that there is a lot of knowledge to learn from this transition.

- To test more approaches from the subfield of Safe RL presented in chapter 5. We already took the idea of LfD methods from this chapter, but we would like to couple safety approaches with the SAC algorithm. These methods could add safety and stability guarantees to the learning process. For the example, the *Lyapunov Neural Networks (LNN)* described in section 5.2.3 seems to be a promising algorithm. We could for example compare the Lyapunov functions estimated by the LNNs with the ones estimated by the computational methods mentioned in section 2.3.4.

- To experiment the advanced versions of the SAC algorithm mentioned at the end of section 3.2.3.3. These new versions designed by the same authors as in [116] seemed to improve the performance of SAC on several tasks. We would like to see if the changes made inside the algorithm can impact the performance of our DRL-based controller for the AUV waypoint tracking task.

From a long-term perspective, our work falls within the project of facilitating the control of AUVs thanks to the use of ML. The proposals made here represent a first step towards this goal and allow to identify the most important aspects to handle during the implementation of RL algorithms for the end-to-end control of AUV. We think that this goal can be achieved thanks to the progressive integration of elements taken from the control theory or robotics inside the ML methods. We would like to have the best of both worlds: the adaptability and the innovative control approaches achieved by the ML, as well as the guarantees and the knowledge provided by the control theory and robotics. A lot of the RL methods presented in chapter 5 are based on elements from the control theory and show that this unification is possible. The integration of expert knowledge in ML could result in the creation of hybrid methods.

Appendix A

Additional elements of traditional machine learning

A.1 Traditional machine learning

As explained in the introduction, there was an era called the traditional machine learning before the emergence of deep learning approaches. Many well-known algorithms were developed at this time for supervised and unsupervised learning tasks: k-nearest neighbors, k-means, support vector machine, decision trees, random forest, etc. In this appendix, we are going to described the major elements of traditional machine learning, which have been later reused by deep learning and reinforcement learning algorithms. We will focus our presentation on supervised learning tasks, because similar ideas are often found in deep reinforcement learning algorithms like the SAC. However, a few works mixing unsupervised learning and reinforcement learning can be found in the literature [153][379], but they represent a minority of the tasks so they will not be presented here. Finally, the books [36], [124] and [154] are three useful resources allowing researchers to have an exhaustive point of view on the field of traditional machine learning.

A.1.1 Preprocessing

In traditional machine learning, a preprocessing step needs to be performed before beginning the training phase. The raw data are prepared in order to extract the most useful features from them. These extracted features will constitute the inputs of the machine learning algorithm, instead of the raw data.
This preprocessing step is carried out using technical knowledge from the application domain of the task: image processing, signal processing, statistics, etc. An expert from the given application domain is needed in order to choose the most representative features with respect to the type of dataset and the type of task, so as to achieve the best possible performance. This

expert can be a different person from the data scientist, allowing to separate the application domain work and the machine learning work.
This preprocessing step is no longer found in the deep learning approaches, where the raw data are directly sent to the input layers of neural networks (see the section 3.1.1).

A.1.2 Training set, development set and test set

In supervised learning tasks, the initial dataset is always divided into several independent subsets:

- The training set: the labeled data contained in the training set are used to carry out the training phase of the algorithm. This is the phase where the learning process occurs, by updating the parameters in order to fit the data.

- The development set: The data found in the development set allows to find the right hyperparameters, by comparing different configurations of a same algorithm. Each version of the chosen algorithm is initialized with different values of hyperparameters, before training all versions on the same training set.
  After that, their performance are evaluated on the data of the development set. It is the first time that the algorithms encounter these specific data, since the data from the development set are not used during the training phase. These data represent unknown data from the point of view of the algorithm and allows to evaluate their ability to generalize their predictions to new data. The learning is here disabled, which means that the parameters of each algorithm are fixed.

- The test set: the role of the test set is to make an assessment of the final performance of a trained algorithm. After the training and choosing the hyperparameters, the performance of the final algorithm is assessed on new data that were never been encountered during the training of the algorithm and the choice of its hyperparameters, in to have an unbiased evaluation.

The initial dataset must be splitted randomly into these three subsets, and the class distribution over data inside each subset must be the same as inside the initial dataset. This is called stratified sampling [297]. When the amount of data is inferior to 100 000 examples, the distribution between the subsets is generally 60% for the training set, 20% for the development set and 20% for the test set. When it is above 100 000 examples, the repartition of data become something close to 98% for the training set, 2% for the development set and 2% for the test set. 2% of a very large amount of data is still sufficient to assess correctly the performance of the algorithms, which allows to keep a maximum of data for the proper learning of the task.
Ither advanced methods of performance evaluation can be found in the literature, such that the cross-validation method for example [291][27].

		Predicted Class	
		Spam	No-spam
Actual Class	Spam	12	3
	No-spam	4	81

Figure A.1. An example of confusion matrix for a spam recognition task.

A.1.3 Assessing the performance with the good metrics

Each machine learning task needs its own metrics in order to evaluate the performance of the learning algorithm on each subsets [53]. In supervised learning, the most common representation of the performance evaluation is the *confusion matrix* [223][343]. An example of a confusion matrix is given in Figure A.1 [1], in the case of a binary classification task of spam recognition used in emailing system. This representation sums up different aspects of the performance of a given algorithm, according to the definitions shown on the Figure A.2 (taken from [270]). Depending on the nature of the task, these aspects will have different importance. For example for a critical system such as a military radar, minimizing the number of false negatives becomes more important than minimizing the number of false positives: it is more important to not miss a true target (a false negatives), than to eventually signal nonexistent targets (false positives). The false positives can be manually invalidated afterwards.
In multiclass classification task, the size of the confusion matrix increases in order to show all the possible combinations of classes regarding the actual and predicted classes. The Figure A.3 [2] shows the confusion matrix of a handwritten digit recognition task, expressed in percentages.

Several metrics can then be compute based on the different components of the confusion matrix (we kept the notation of the Figure A.2):

- $Precision = \frac{TP}{TP+FP}$
- $Recall = \frac{TP}{TP+FN}$
- $Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$
- $F1\ score = \frac{2}{\frac{1}{Precision}+\frac{1}{Recall}}$

[1] Taken from *What Is a Confusion Matrix?*, Aman Goel, https://magoosh.com/data-science/what-is-a-confusion-matrix/

[2] Taken from *Recognizing hand-written digits*, Official scikit-learn documentation, https://scikit-learn.org/stable/auto_examples/classification/plot_digits_classification.html

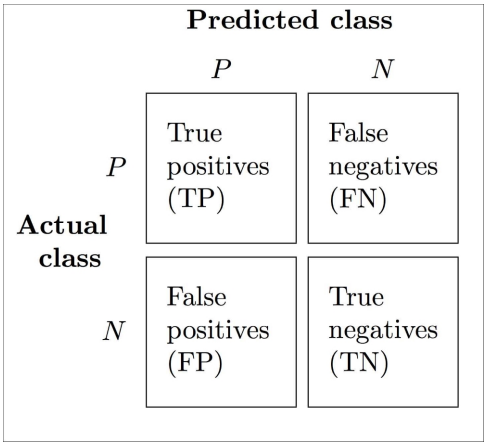


Figure A.2. The definitions of the components of a general confusion matrix.

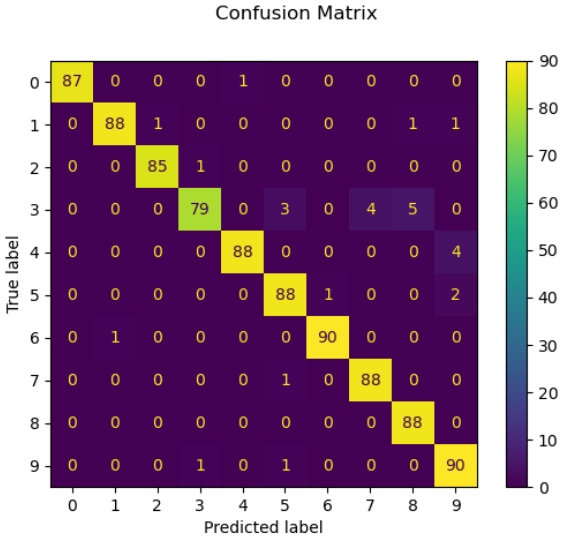


Figure A.3. The confusion matrix of a handwritten digit recognition task, in percentages.

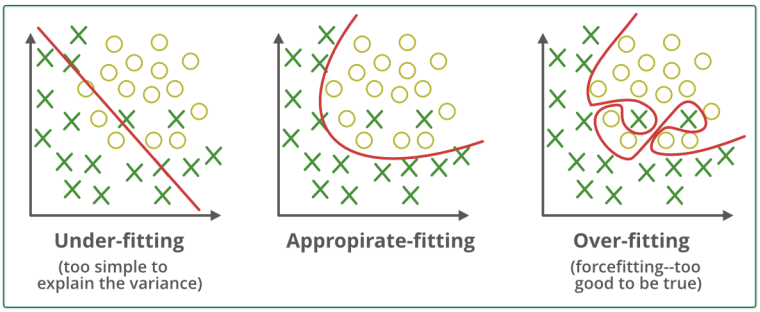


Figure A.4. Overfitting and underfitting for a binary classification task example.

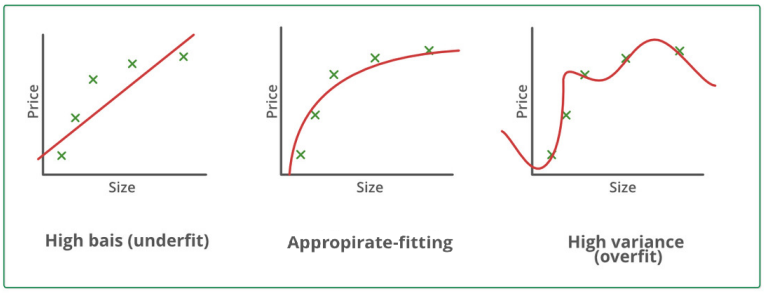


Figure A.5. Overfitting and underfitting for a regression task example.

The simultaneous use of several metrics during the training process of a model can help the data scientist to choose the right hyperparameters and to choose among different approaches.

A.1.4 Overfitting and underfitting

Overfitting and underfitting are concepts used in supervised learning in order to described how well the machine learning algorithm managed to fit the training data [152][370].
In statistics, overfitting means that a statistical analysis or a model is too close from its training data, which will result in the model not fitting new data. Overfitting prevents the model from correctly predicting data coming from the development set and the test set. The model is mapping too closely the data from the training set, even if outliers can be found inside.
At the opposite, underfitting means that the model is not fitting enough the training data, and is not correctly learning the task. It is not able to retrieve the correct labels of neither the training data, nor the development/test data.

The Figure A.4 [3] shows an example of a binary classification task: the algorithm has to

[3] Taken from *Underfitting and Overfitting*, https://www.geeksforgeeks.org/underfitting-and-overfitting-in-machine-learning/

classify the data as *cross* or a *circle*. If the model is underfitting, the decision boundary allowing to differentiate the two classes is coarse and not precise enough. In the case of overfitting, the decision boundary is too close to the outliers (the two crosses found among the circles are outliers, they should not be labeled as *cross*). In the case of an appropriate fitting, the model is able to recognize the outliers and to rectify their classes (it classifies the crosses as circles, which is what would be expected at these two locations).
The Figure A.5 [3] shows an example of a regression task: the algorithm has to estimate a function (the red line) fitting several training data (the green crosses). In the case of underfitting, the estimated function is a straight line that is not following the shape drawn by the data points. In the case overfitting, the model is taking too much liberty and the estimated function is composed of unrealistic curvatures. In the case of an appropriate fitting, the model is able to produce a function fitting the data point locations.

Specific statistical terms can be employed to further analyse the algorithm behaviour: if the model is underfitting, the algorithm is said to have high bias and low variance; if it is overfitting, it has high variance and low bias. During each machine learning task, a trade-off must be found between the bias and the variance. In traditional machine learning, the final model cannot have simultaneously low bias and low variance. The deep learning approaches make it possible to get away from this trade-off and allow to produce low-bias-low-variance models.
Very often, a model is underfitting because it is too simple for the learning task. An algorithm with more modelling abilities is then needed, including more nonlinearities. At the contrary, a model can overfit because it is way too complex for the task. Overfitting can also happen because there are not enough training data. Moreover, if several classes are not well represented in the dataset, more impact can be give to potential outliers or inconsistent data.

A.1.5 Logistic regression

The logistic regression [177][137] is a well-known traditional machine learning algorithm and will be presented here. It is a fundamental building block of deep learning, since each neurons found in a neural network can be viewed as a modified logistic regression (see section 3.1). Logistic regression is used for binary classification tasks in the subfield of supervised learning.

A.1.5.1 Inputs and outputs

The input received by the logistic regression algorithm is composed of multiple vectors $\mathbf{x_i}$, where $i \in [1, m]$. Each $\mathbf{x_i}$ is a data sample taken from the training set, the development set or the test set (depending of which stage of the training process is carried out), and all these vectors are stacked in order to form the input matrix $\mathbf{X}$. A label y_i is assigned to each input $\mathbf{x_i}$ in order to form a pair $(\mathbf{x_i}, y_i)$. The label y_i is a scalar representing the number of the class

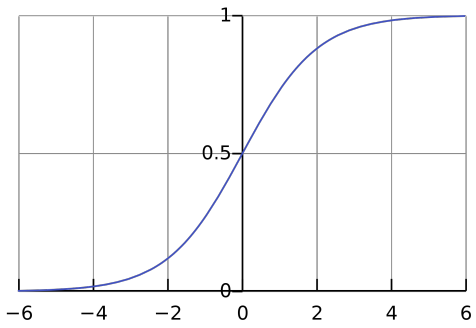


Figure A.6. The sigmoid function

to which the sample $\mathbf{x_i}$ belongs. All the labels y_i can also be stacked together in order to form a vector $\mathbf{y}$. The use of matrices and vectors allows to speed up the computations by applying linear algebra operations. The computations can be performed on all the training samples together, instead of being handled individually by the algorithm.

The output of the logistic regression is defined as a real scalar ranging from 0 to 1, called $\hat{y_i}$, and corresponding to an estimation of the true label y_i assigned to the input $\mathbf{x_i}$. We can scaled this output interval in order to fit the range of y_i, by multiplying the output by a factor and adding a real value.
If the use case is a binary classification task, a threshold is used in order to differentiate the two classes:

$$\begin{cases} \textbf{if } \hat{y_i} < 0.5 \quad \textbf{then} \quad \hat{y_i} = class\ 0 \\ \textbf{if } \hat{y_i} \geq 0.5 \quad \textbf{then} \quad \hat{y_i} = class\ 1 \end{cases} \tag{A.1}$$

A logistic regression has only two parameters that are updated during the training process: a vector $\mathbf{w}$ called the *weight* and a scalar b called the *bias*.
The output $\hat{y_i}$ is computed from a given input $\mathbf{x_i}$ using the following equation:

$$\hat{y_i} = \text{sigmoid}\left(\mathbf{w}^T \mathbf{x_i} + b\right) \tag{A.2}$$

$$\text{with } \text{sigmoid}(z) = \frac{1}{1 + \mathrm{e}^{-z}} \tag{A.3}$$

An example of *sigmoid* function is plotted on the Figure A.6 [4]. This function allows to keep the value of the output $\hat{y_i}$ inside the interval [0,1]. Moreover, this function is nonlinear

[4]Taken from *Logistic regression*, https://en.wikipedia.org/wiki/Logistic_regression

and adds some complexity in the decision making process of the logistic regression, compared to a linear regression for example. This ability is illustrated on the figures A.7 (taken from [80]) and A.8 [5], which represent respectively the decision boundaries of a linear regression and a logistic regression on a binary classification task. We can see that the decision boundary of the logistic regression have a form more complex than the one of the linear regression

In practice, the previous equations are adapted in order to be able to compute the estimated labels $\hat{y_i}$ of several input data $\mathbf{x_i}$ at the same time, using linear algebra calculation rules and vectorization. This allows to recude the computational time.

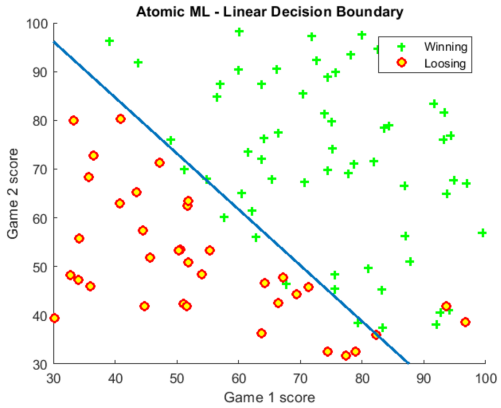


Figure A.7. The decision boundary generated by a linear regression on a binary classification task.

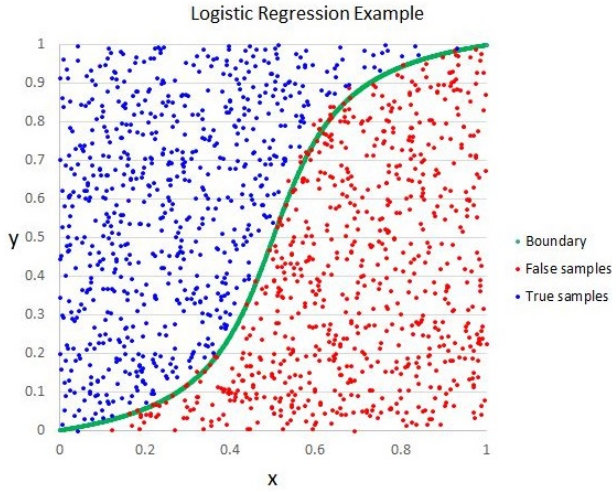


Figure A.8. The decision boundary generated by a logistic regression on a binary classification task.

[5]Taken from *Basics of Classification Models*, https://antoniohila.medium.com/basics-of-classification-models-20487d4d6498

A.1.5.2 Loss function

Once all the $\hat{y}_i$ have been estimated, the learning process needs an error measure in order to figure out the classification or regression performance of the machine learning algorithm. This performance measure is called the *loss function* L [314][156] when the performance is assessed on only one example and the *cost function* J when it is done on a whole set of data. The goal of the algorithm is to update its parameters in order to minimize the loss and the cost functions.

In the case of the logistic regression, the loss function is defined as follows:

$$L(\hat{y}_i, y_i) = -\left(y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)\right) \tag{A.4}$$

It is called the *cross-entropy* loss function [241] and is a very well-known loss function in the ML community. We are going to explain the utility of the cross-entropy loss function for binary classification tasks:

- If the true label y_i of the input data x_i is equal to 0, the equation A.4 becomes:

$$L(\hat{y}_i, 0) = -\log(1 - \hat{y}_i) \tag{A.5}$$

  In this case, minimizing the loss function is equivalent to maximizing the term $\log(1-\hat{y}_i)$, and consequently the term $(1 - \hat{y}_i)$. This means that when the algorithm is minimizing the loss function, it is minimizing the estimated label $\hat{y}_i$. Given that $\hat{y}_i$ is composed of a sigmoid, as defined in the equation A.2, the estimated label tends to 0 when it is minimized. By applying a threshold as in A.1, the algorithm ends up by classifying the input as belonging to the correct class 0.

- If the true label y_i of the input data x_i is equal to 1, the equation A.4 becomes:

$$L(\hat{y}_i, 1) = -\log(\hat{y}_i) \tag{A.6}$$

  By following the same reasoning, minimizing the loss function is equivalent to maximizing the term $\log(\hat{y}_i)$, and consequently the estimated label $\hat{y}_i$. Given that $\hat{y}_i$ is composed of a sigmoid, as defined in the equation A.2, the estimated label tends to 1 when it is maximized. By applying a threshold as in A.1, the algorithm ends up by classifying the input as belonging to the correct class 1.

A.1.5.3 Cost function

The cost function is simply defined as the mean loss function over the set of all data inputs, for a given configuration of the parameters of the logistic regression. For a given weight $\mathbf{w}$, a given bias b and a given set of input data $\mathbf{x_i}$ composed of n samples, the cost function J is defined as:

$$J(\mathbf{w}, b) = \frac{1}{m} \sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right) \tag{A.7}$$

The equation A.7 is valid for all possible definitions of the loss function cited previously.

A.1.5.4 Gradient descent

In ML, he training process consist of updating the parameters of the learning algorithm in order to fit the training data. In the case of the logistic regression, the weight $\mathbf{w}$ and the bias b must be updated in order to minimize the cost function A.7. Indeed, the cost function is an error measure reflecting how a given model is fitting the dataset by measuring the differences between the true labels and the predicted labels of all the data.

The gradient descent is used in order to update of the parameters of the logistics regression:

$$\begin{cases} \mathbf{w} = \mathbf{w} - \alpha \frac{\partial J(\mathbf{w},b)}{\partial \mathbf{w}} \\ b = b - \alpha \frac{\partial J(\mathbf{w},b)}{\partial b} \end{cases} \tag{A.8}$$

where α is a real scalar called the *learning rate*.

The weight $\mathbf{w}$ and the bias b are updated in the equation A.8 in order to follows the direction given by the partial derivative of the cost function at a given point in the space of the parameters. The sign minus in front of the learning rate indicates that the parameters are updated in order to move towards the minimum of the cost function.

The learning rate α is a hyperparameter allowing to adjust the steps between each parameter updates, in order to control the speed and the precision of the convergence towards the minimum of the cost function. To summarize, the gradient descent updates the parameters proportionally to the errors made in the estimates of the true labels.

Appendix B

Additional elements of Deep learning

This appendix will described additional techniques used in practice in the deep larning community: the initialization and the regularization of the neural networks, as well as other optimization algoritms.

B.1 Regularization

As explained A.1.4, overfitting can sometime occur during the training of Machine Learning algorithms. This will lead the neural networks too stay "too close" to the training data, and they will not be able to generalize well to new data: they will have a high variance.

The *regularization* techniques were created in order to avoid overfitting as much as possible. One of the most known regularization technique is the *L2 regularization* [61][340]. The idea of this method is to reduce the magnitude of the weights $\mathbf{W}$ of the neural network.
The L2 regularization simply redefined the cost function as:

$$J(\mathbf{W}, \mathbf{b}) = \frac{1}{m} \sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2m} ||\mathbf{W}||_2{}^2 \tag{B.1}$$

where λ is a hyperparameter alowing to control the trade-off between minimizing the classification error and the norm of the weights parameters.

In this way, the L2 norm of the weights $\mathbf{W}$ is minimized at the same time as the classic loss function, and it prevents the weights $\mathbf{W}$ too explode to extreme magnitude.
For this reason, the L2 regularization is sometimes called the *weight decay* technique.

Other regularization techniques exist [61][250]: the L1 regularization [186], the dropout regularization [347], the early stopping [267], the data augmentation [132], the model ensembles

[57], etc.

B.2 Initialization of the neural networks

As the training process progresses, the magnitude of the parameters update can become very small, or very large. This will leads the gradients to explode or to vanish: they become so large or so small, that they will interfere with the training process, and the gradient descent will take too small steps, or too large steps.
In order to avoid that, a special initialization method is needed for the weights of the neural network.

The *He initialization* method [126] initializes the weight matrix $\mathbf{W_i}$ of the layer i by randomly initialize its values using a standard normal distribution with a variance σ^2 defined as:

$$\sigma^2 = \frac{1}{n^{[i-1]}} \tag{B.2}$$

where $n^{[i-1]}$ is the number of neurons found in the layer $(i-1)$.

The *Xavier initialization* method [106] initializes the weight matrix $\mathbf{W_i}$ of the layer i by randomly initialize its values using a standard normal distribution with a variance σ^2 defined as:

$$\sigma^2 = \frac{2}{n^{[i-1]} + n^{[i]}} \tag{B.3}$$

The idea of these two initialization methods is to scale the magnitude of the weights of a given layer according to the number of neurons found in this layer and in the previous layer.
The bias $\boldsymbol{b}_i$ are initialized to zero.

B.3 Optimizers

The Adam optimizer described in the section 3.1.2.2 is the most common optimizer found in the literature, but other optimizers exists. All of the following algorithms are evolutions of the basic gradient descent.

B.3.1 Mini-batch gradient descent

As explained in the section 3.1, neural networks require very large amounts of training data in order to perform well. When the gradient descent is applied to neural networks during the

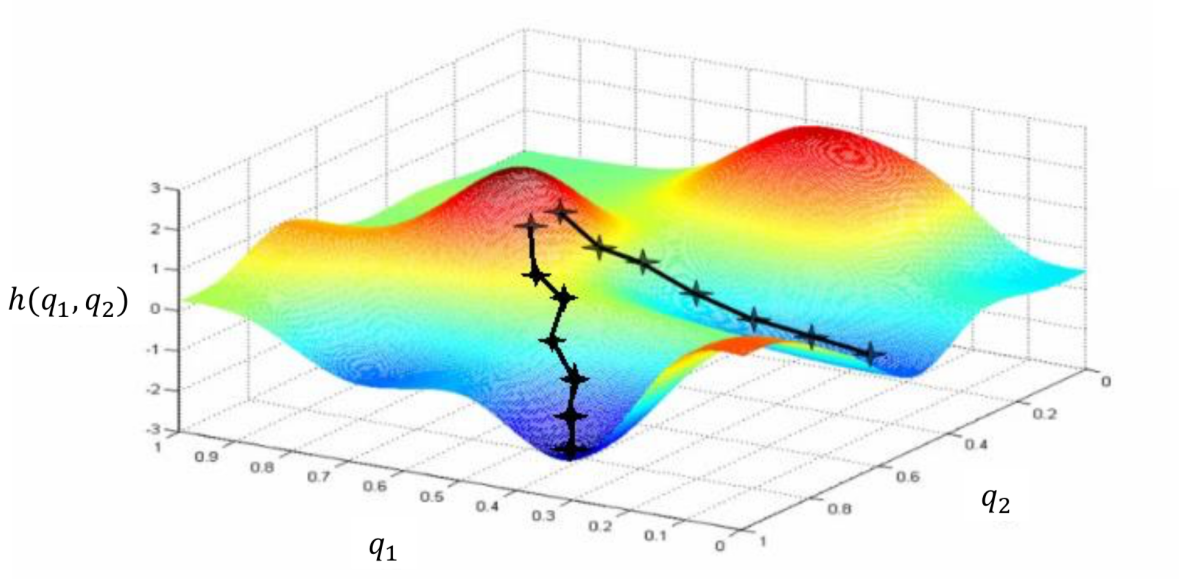


Figure B.1. An example of a gradient descent algorithm. It is updating two parameters q_1 and q_2 in order to minimize a function $h(q_1, q_2)$.

backpropagation (as defined in 3.1.1), the more data are involved, the more computation time it will take to perform one step of gradient descent.

We recall that the goal of the gradient descent algorithm is to seek the minimum of the cost function $J(\mathbf{W}, \mathbf{b})$. It starts from an initial point $(\mathbf{W_0}, \mathbf{b_0})$ and modifies the parameters $\mathbf{W}$ and $\mathbf{b}$ in order to move towards the local minimum of the function, by taking the direction given by the gradients $\frac{\partial J(\mathbf{W},\mathbf{b})}{\partial \mathbf{W}}$ and $\frac{\partial J(\mathbf{W},\mathbf{b}))}{\partial \mathbf{b}}$. The Figure B.1 [1] shows an example of a gradient descent algorithm.

The principle of *mini-batch gradient descent* [279] is to divide the training dataset into smaller datasets called *mini-batch*, and to perform a step of gradient descent on each mini-batch independently.
Since the gradient descent is performed on a smaller amount of data samples, less computations are necessary and one step of gradient descent is faster, reducing the time taken by the deep learning algorithm to converge.
The downside of this is that, because of the fact that less data samples are used to compute the gradients during the backpropagation, the gradient descent is more sensible to ouliers

[1] Taken from *01 and 02: Introduction, Regression Analysis, and Gradient Descent*, http://www.holehouse.org/mlclass/$01_02 Introduction_r egression_a nalysis_a nd_g r.html$

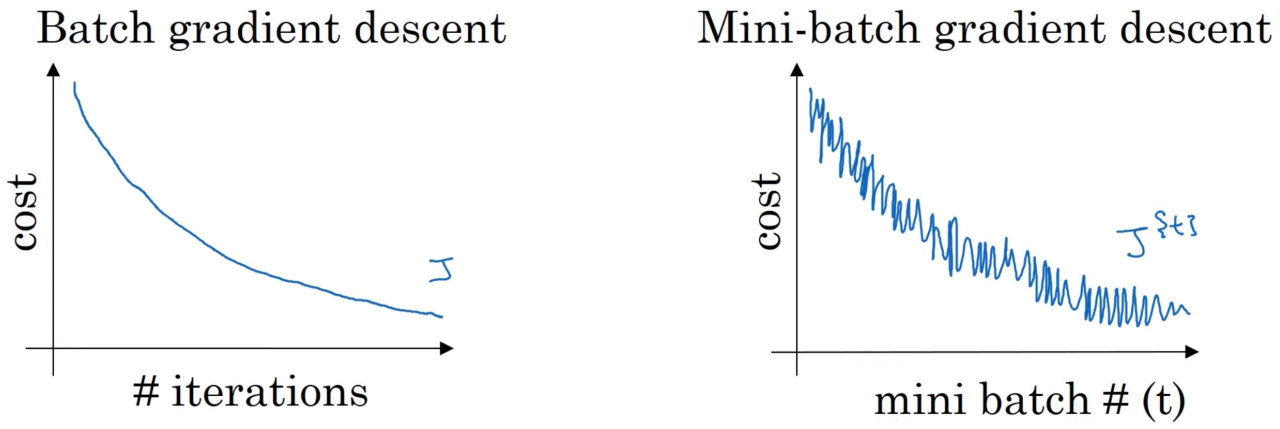


Figure B.2. Illustration of the impact of the mini-batch gradient descent on the loss function of a neural network.

data (similarly to the outliers discussed in section A.1.4): these outliers will indicate a wrong information and are considered as an error in the dataset. The gradient computations applied to outliers data will not give a correct direction and will not make the gradient descent converge towards the minimum of the cost function. The less data are taken in mini-batches, the more these incorrect outliers will have weight in the final direction taken by the gradient descent in the parameter space.
This effect is illustrated on the Figure B.2 [2], where the cost function appears to be noisier during the use of mini-batch gradient descent. This noise come from the outliers found in the mini-batches, making the neural network predict wrong labels and causing the loss function to not decrease monotonically.

Once the gradient descent has been performed on all the different mini-batches, we say that one *epoch* has been performed. The number of epochs and the size of the mini-batches are two additional hyperparameters which need to be tuned.
A practical rule concerning the size of the mini-batches is that it has to be a power of 2, similarly to the way the computer memory is constructed.

B.3.2 Gradient descent with momentum

As explained previously, the approach of training a neural network on less data samples by using mini-batches makes the gradient descent more sensitive to outliers. This causes the loss function to not decrease monotonically, and the gradient descent to not modify the parameters of the neural network in the correct way: some steps of the backpropagation (that is supposed

[2]Taken from *Improving Deep Neural Networks: Hyperparameter Tuning, Regularization and Optimization*, https://www.coursera.org/learn/deep-neural-network?specialization=deep-learning

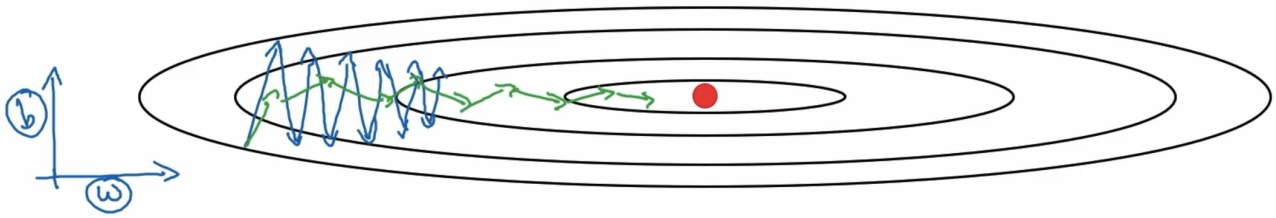


Figure B.3. Illustration of the impact of the gradient descent with momentum algorithm on the convergence towards the minimum of the cost function. Blue: mini-batch gradient descent; green: gradient descent with momentum.

to minimize the loss function as explained in section) do not allow the loss function to converge directly to its minimum.
The algorithm called *gradient descent with momentum* is an improvement of the mini-batch gradient descent aiming to make the loss function converge more directly towards the mininmum. It first appeared in [280] and is usually combined with mini-batch gradient descent.

The idea of gradient descent with momentum is to update the parameters of the neural network by using a *moving average* of the partial derivatives of the loss function, instead of directly using the raw partial derivatives.
During the training of a given neural network, the equations 3.4 are replaced by:

$$\begin{cases} \mathrm{MA}_{\mathbf{W_i}} = \beta\ \mathrm{MA}_{\mathbf{W_i}} + (1-\beta)\ \frac{\partial J(\mathbf{W_i},\mathbf{b_i})}{\partial \mathbf{W_i}} \\ \mathrm{MA}_{\mathbf{b_i}} = \beta\ \mathrm{MA}_{\mathbf{b_i}} + (1-\beta)\ \frac{\partial J(\mathbf{W_i},\mathbf{b_i})}{\partial \mathbf{b_i}} \\ \mathbf{W_i} = \mathbf{W_i} - \alpha\ \mathrm{MA}_{\mathbf{W_i}} \\ \mathbf{b_i} = \mathbf{b_i} - \alpha\ \mathrm{MA}_{\mathbf{b_i}} \end{cases} \quad \forall\, i \in [1, N] \tag{B.4}$$

where MA stands for *Moving Average* and is implementing an exponentially weighted moving average of the partial derivative of the cost function J with respect to both parameter vectors $\mathbf{W_i}$ and $\mathbf{b_i}$ of the *i-th* layer of the neural network. β is a hyperparameter included in the interval [0,1], α is the learning rate and N is the number of layers of the neural network.
The hyperparameter β is often set to 0.9 in the deep learning community, and allows to define an exponentially weighted moving average taken for the last 10 samples: by definition, an exponentially weighted moving average with a given hyperparameter β is operating an average on the last $\frac{1}{1-\beta}$ entries.
This operation is repeated for each mini-batches of each epoch.

The Figure B.3 [3] illustrates the impact of the gradient descent with momentum, compared to the mini-batch gradient descent. We can see that the parameters are updated in order to converge more directly towards the minimum of the loss function. The gradient descent steps are smoothed by the gradient descent with momentum, due to the fact that one step is now the result of an average on multiple partial derivatives. Less weight is given to the outliers.

B.3.3 RMSprop

RMSprop stands for *Root Mean Square Propagation* and is an optimizer which was first found in [134]. It is very similar to gradient descent with momentum. Its purpose is also to make the convergence of the gradient descent faster and to smooth the steps taken in the parameter space towards the minimum of the loss function.

The gradient descent update equations 3.4 become:

$$\begin{cases} \text{SMA}_{\mathbf{W_i}} = \beta\ \text{SMA}_{\mathbf{W_i}} + (1-\beta)\left(\frac{\partial J(\mathbf{W_i},\mathbf{b_i})}{\partial \mathbf{W_i}}\right)^2 \\ \text{SMA}_{\mathbf{b_i}} = \beta\ \text{SMA}_{\mathbf{b_i}} + (1-\beta)\left(\frac{\partial J(\mathbf{W_i},\mathbf{b_i})}{\partial \mathbf{b_i}}\right)^2 \\ \mathbf{W_i} = \mathbf{W_i} - \alpha\ \frac{\frac{\partial J(\mathbf{W_i},\mathbf{b_i})}{\partial \mathbf{W_i}}}{\sqrt{\text{SMA}_{\mathbf{W_i}}+\epsilon}} \\ \mathbf{b_i} = \mathbf{b_i} - \alpha\ \frac{\frac{\partial J(\mathbf{W_i},\mathbf{b_i})}{\partial \mathbf{b_i}}}{\sqrt{\text{SMA}_{\mathbf{b_i}}+\epsilon}} \end{cases} \quad \forall i \in [1,N] \tag{B.5}$$

where SMA stands for *Squared Moving Average* and is implementing an exponentially weighted moving average of the square of the partial derivative of the cost function J with respect to the parameter vectors $\mathbf{W_i}$ and $\mathbf{b_i}$ of the *i-th* layer of the neural network. β is a hyperparameter included in the interval [0,1], α is the learning rate, ϵ is a small real number avoiding the denominators of the fractions to be equal to zero, and N is the number of layers of the neural network.
Like previously, β is set to 0.9 in order to operate a moving average on the last 10 samples. ϵ can be set to 10^{-8}. The squaring operation is applied element-wise to the vectors of the partial derivatives.

The goal of gradient descent with momentum was to converge faster by taking larger steps of gradient descent. This was achieved by taking the mean of the directions suggested by

[3] Taken from *Improving Deep Neural Networks: Hyperparameter Tuning, Regularization and Optimization*, https://www.coursera.org/learn/deep-neural-network?specialization=deep-learning

the partial derivatives of the loss function. With RMSprop the goal of taking larger steps in order to converge faster is the same, but the way to achieve it is different: the variance or the oscillations of the steps are reduced, in order to make them larger. The oscillations of the steps (which are induced by the transverse directions given by specific partial derivative values) are reduced by normalizing the partial derivatives with the square root of the mean of the square of themselves.

Appendix C

Additional elements of Reinforcement learning

In this appendix, we are going to detail several algorithms mentioned in the section 3.2,which will belong to either the Temporal-Difference learning approach or the policy gradient approach.

C.1 Temporal-Difference learning

Temporal-Difference (TD) learning is a family of algorithms based on the TD errors (defined in the section 3.2.2.1). SARSA and Q-leanring are two basic TD learning algorithm on which the DQN algorithm is based (described in the section 3.2.2.3).

C.1.1 SARSA

SARSA [281] is the name of a classic TD learning algorithm. Its name comes from the typical sequence of operations followed by an agent: $\ldots, S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}, \ldots$

The pseudocode of the SARSA algorithm is (given for only one episode for this example):

---

1. Initialization of $t = 0$.
2. The agent starts with the state S_0 and chooses the action $A_0 = \arg\max_{a \in \mathcal{A}} Q(S_0, a)$, by applying the ϵ-greedy exploration strategy.
3. At time t, after applying action A_t, it observes the reward R_{t+1} and gets into the next state S_{t+1}.

4. It pick the next action in the same way as in step 2, by applying the ϵ-greedy strategy: $A_{t+1} = \arg\max_{a \in \mathcal{A}} Q(S_{t+1}, a)$

5. The Q-value function is updated using the TD error:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)) \tag{C.1}$$

6. The time step is set to $t = t + 1$ and the loop is repeated from step 3, until reaching a pre-defined time step T.

Once the Q-value function is considered to be approximated enough, the policy is derived by acting greedily with respect to the Q-values: by taking the action leading to the maximum Q-value in each state visited by the agent.

It is important to note that in the step 5, the Q-value function is updated using the TD target $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$, which is computed using the same action A_{t+1} taken in the step 4 by the exploration policy.
We say that it is an *on-policy* algorithm: the policy used to compute the TD target (the *target policy*) is the same policy used by the agent to explore the environment (the *exploration policy*) .

The SARSA algorithm is a model-free, value-based, on-policy TD learning algorithm.

C.1.2 Q-learning

Q-learning [355] is a TD learning algorithm very similar to SARSA.

Its pseudocode is (given for only one episode for this example):

---

1. Initialization of $t = 0$.

2. The agent starts with the state S_0.

3. At time t, the agent chooses the action $A_t = \arg\max_{a \in \mathcal{A}} Q(S_t, a)$, by applying the ϵ-greedy exploration strategy.

4. After applying action A_t, it observes the reward R_{t+1} and gets into the next state S_{t+1}.

5. The Q-value function is updated using the TD error:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_{a \in \mathcal{A}} Q(S_{t+1}, a) - Q(S_t, A_t)) \tag{C.2}$$

6. The time step is set to $t = t + 1$ and the loop is repeated from step 3, until reaching a pre-defined time step T.

Again, the final policy used during the deployment of the agent is derived by acting greedily with respect to these estimated Q-values.

This time, the action used to compute the TD target is not choose by the exploration policy (which applies an ϵ-greedy strategy), but is simply the action leading to the maximum Q-value found in the state S_{t+1} (it is the plain greedy policy). This action could be different from the action A_{t+1} that will be selected by the exploration policy in the next state.
The algorithm is said to be *off-policy*: the policy used to compute the TD target (the *target policy*) and learn the Q-values is different from the policy used in the exploration process (the *exploration policy*).

The Q-learning algorithm is a model-free, value-based, off-policy TD learning algorithm.

The opposition between on-policy and off-policy approaches is a key concept that will be encountered with the state-of-the-art algorithms presented in the following sections.

C.2 Policy gradient

The polic-gradient (PG) algorithms presented here have greatly influenced the SAC algorithm and are very popular among the deep RL community.

C.2.1 Deterministic policy gradient

In the PG methods presented in the section 3.2.3, the policy was always stochastic, meaning that $\pi(.|s)$ was a probability distribution over the possible actions $\mathcal{A}$, given the current state. In some algorithms, the policy becomes deterministic and the same action will be always chosen for a given state (until the policy is updated). In the deterministic cases, the policy will be noted μ and we will have $a = \mu(s)$.

Different notations will be used specifically for Deterministic Policy Gradient (DPG) algorithms:

- $\rho_0(s)$: it is the initial distribution over states.
- $\rho_\mu(s \to s', k)$: starting from state s, it the visitation probability density at state s' after moving k steps by following policy μ.
- $\rho_\mu(s')$: it is the marginal of the state distribution induced by the policy μ and is defined as:

$$\rho_\mu(s') = \int_{\mathcal{S}} \sum_{k=1}^{\infty} \gamma^{k-1} \rho_0(s) \rho_\mu(s \to s', k) ds \tag{C.3}$$

  where γ is the same discount factor as in the section 3.2.1.

In the case of DPG agents, the objective function 3.27 becomes:

$$J(\boldsymbol{\theta}) = \int_{\mathcal{S}} \rho_\mu(s) Q(s, \mu_{\boldsymbol{\theta}}(s)) ds \tag{C.4}$$

Applying the chain rule to the gradients calculus, the policy gradient theorem 3.30 becomes:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \int_{\mathcal{S}} \rho_\mu(s) \nabla_a Q^\mu(s, a) \nabla_{\boldsymbol{\theta}} \mu_{\boldsymbol{\theta}}(s)|_{a=\mu_{\boldsymbol{\theta}}(s)} ds \\ &= \mathbb{E}_{s \sim \rho_\mu}[\nabla_a Q^\mu(s, a) \nabla_{\boldsymbol{\theta}} \mu_{\boldsymbol{\theta}}(s)|_{a=\mu_{\boldsymbol{\theta}}(s)}] \end{aligned} \tag{C.5}$$

It is called the *deterministic policy gradient theorem* [306].

C.2.2 Deep deterministic policy gradient

Deep Deterministic Policy Gradient (DDPG) [212] is one of the most used deep RL algorithm in the literature. It is an off-policy PG algorithm based in an Actor-Critic architecture, and combining the DPG [306] and the DQN [229] algorithms.

It implements an Actor-Critic architecture, but here both the policy and the value function are modeled by a neural network, like for the value function found in the DQN algorithm (see section 3.2.2).
The actor models a deterministic policy $\mu(s|\boldsymbol{\theta}^\mu)$ using a neural network of parameters $\boldsymbol{\theta}^\mu$. The neural network parameters of the actor are updated using the backpropagation with the deterministic policy gradient theorem C.5.
The critic models the action-value function $Q(s, a|\boldsymbol{\theta}^Q)$ using a neural network of parameters $\boldsymbol{\theta}^Q$. The neural network parameters of the critic are updated using the backpropagation with the computation of TD errors.

Like the DQN algorithm, both the actor and the critic have also target neural networks used for computing the TD errors of the action-value function. The DDPG has four neural networks: the policy network μ, the target policy network μ', the Q-value network Q and the target Q-value network Q'.
These target networks help to stabilize the learning of the Q-function, but also indirectly the policy (since the Q-values are found in the deterministic policy gradient theorem).
Instead of being kept frozen for a certain number of steps before being updated (as in the DQN), here the parameters of the target neural networks are slowly updated in a continuous way, using an exponential moving average:

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta' \qquad \text{with } \tau \ll 1 \tag{C.6}$$

It constrains the target network values to change slowly. We say that the DQN is performing a *hard update*, whereas the DDPG is performing a *soft update*.

The DDPG algorithm is off-policy, meaning that the exploration policy (the policy collecting the training samples) and the target policy (the policy used in the TD error) are different.
The exploration policy $\mu_{\text{exp}}(s)$ used by the DDPG is constructed by adding noise to the current policy $\mu(s|\boldsymbol{\theta}^{\mu})$

$$\mu_{\text{exp}}(s) = \mu(s|\boldsymbol{\theta}^{\mu}) + \mathcal{N} \tag{C.7}$$

where $\mathcal{N}$ is a random process.

Like the DQN, DDPG implements the mechanism of experience replay: the policy network and the Q-value network are trained using a batch of training samples taken from a replay buffer D.
Moreover, DDPG uses the batch normalization technique [150] described in the section 3.1.2.3.

Here is the detailed pseudocode of the DDPG alogorithm:

---

1. Random initialization of the parameters $\boldsymbol{\theta}^{\mu}$ and $\boldsymbol{\theta}^{Q}$ of the policy network $\mu(s|\boldsymbol{\theta}^{\mu})$ and the Q-value network $Q(s,a|\boldsymbol{\theta}^{Q})$.
   Initialization of the target networks μ' and Q' as follows: $\boldsymbol{\theta}^{Q'} \leftarrow \boldsymbol{\theta}^{Q}$, $\boldsymbol{\theta}^{\mu'} \leftarrow \boldsymbol{\theta}^{\mu}$
   Initialization of the replay buffer D.
2. **For** episode $= 1, \ldots, M$ **:**
   (a) Initialization of the random process $\mathcal{N}$.
   (b) Initial observation of the state s_1 given by the environment to the agent.

(c) **For** $t = 1, \ldots, T$ **:**

i. Selection of the action a_t by the exploration policy: $a_t = \mu(s_t|\boldsymbol{\theta}^\mu) + \mathcal{N}$

ii. Execution of the action a_t.
Observation of the reward r_t and the next stat s_{t+1} given by the environment to the agent.

iii. The transition (s_t, a_t, r_t, s_{t+1}) is stored in the replay buffer D.

iv. A random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) is sampled from the replay buffer D.

v. Setting of the labels y_i of all training samples using the TD error and the target networks:

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\boldsymbol{\theta}^{\mu'})|\boldsymbol{\theta}^{Q'}) \tag{C.8}$$

Update of the parameters of the Q-value network $Q(s, a|\boldsymbol{\theta}^Q)$ by the critic, using a gradient descent and a backpropagation in order to minimize the loss:

$$L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\boldsymbol{\theta}^Q))^2 \tag{C.9}$$

vi. The policy network $\mu(s|\boldsymbol{\theta}^\mu)$ is updated by the actor using a gradient ascent and a backpropagation in order to maximize the objective function C.4. The deterministic policy gradient theorem allows to compute the gradients needed by the gradient ascent:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\boldsymbol{\theta}^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\boldsymbol{\theta}^\mu)|_{s=s_i} \tag{C.10}$$

vii. Update of the parameters of the target networks using $\tau \ll 1$:

$$\begin{cases} \theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'} \\ \theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'} \end{cases} \tag{C.11}$$

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation*, pages 265–283, 2016.

[2] Pieter Abbeel, Adam Coates, and Andrew Y Ng. Autonomous helicopter aerobatics through apprenticeship learning. *The International Journal of Robotics Research*, 29(13):1608–1639, 2010.

[3] Naoki Abe, Prem Melville, Cezar Pendus, Chandan K Reddy, David L Jensen, Vince P Thomas, James J Bennett, Gary F Anderson, Brent R Cooley, Melissa Kowalczyk, et al. Optimizing debt collections using constrained reinforcement learning. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 75–84, 2010.

[4] Seyed R Ahmadzadeh, Petar Kormushev, and Darwin G Caldwell. Multi-objective reinforcement learning for auv thruster failure recovery. In *2014 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 1–8, 2014.

[5] Zafarali Ahmed, Nicolas Le Roux, Mohammad Norouzi, and Dale Schuurmans. Understanding the impact of entropy on policy optimization. In *International Conference on Machine Learning*, pages 151–160, 2019.

[6] James S Albus, Anthony J Barbera, and Roger N Nagel. *Theory and practice of hierarchical control*. National Bureau of Standards, 1980.

[7] Elie Aljalbout, Vladimir Golkov, Yawar Siddiqui, Maximilian Strobel, and Daniel Cremers. Clustering with deep learning: Taxonomy and new methods. *arXiv preprint arXiv:1801.07648*, 2018.

[8] Brenna D Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and autonomous systems*, 57(5):469–483, 2009.

[9] Benjamin Armstrong, Eric Wolbrecht, and Dean B Edwards. Auv navigation in the presence of a magnetic disturbance with an extended kalman filter. In *OCEANS'10 IEEE SYDNEY*, pages 1–6, 2010.

[10] Mehmet S Arslan, Naoto Fukushima, and Ichiro Hagiwara. Nonlinear optimal control of an auv and its actuator failure compensation. In *2008 10th International Conference on Control, Automation, Robotics and Vision*, pages 668–673, 2008.

[11] Karl J Åström. *Introduction to stochastic control theory*. Courier Corporation, 2012.

[12] Karl J Åström and Tore Hägglund. *Advanced PID control*. ISA, 2006.

[13] Karl J Åström, Tore Hägglund, Chang C Hang, and Weng K Ho. Automatic tuning and adaptation for pid controllers-a survey. *Control Engineering Practice*, 1(4):699–714, 1993.

[14] Karl J Åström and Richard M Murray. *Feedback systems*. Princeton university press, 2010.

[15] Karl J Åström and Björn Wittenmark. *Adaptive control*. Courier Corporation, 2013.

[16] Michael Athans. The role and use of the stochastic linear-quadratic-gaussian problem in control system design. *IEEE transactions on automatic control*, 16(6):529–552, 1971.

[17] Derek P. Atherton and Somanath Majhi. Limitations of pid controllers. In *Proceedings of the 1999 American Control Conference*, volume 6, pages 3843–3847, 1999.

[18] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.

[19] Ying Bai and Dali Wang. Fundamentals of fuzzy logic control—fuzzy sets, fuzzy rules and defuzzifications. In *Advanced fuzzy logic technologies in industrial applications*, pages 17–36. 2006.

[20] Vedran Bakaric, Zotan Vukic, and Radovan Antonic. Improved basic planar algorithm of vehicle guidance through waypoints by the line of sight. In *First International Symposium on Control, Communications and Signal Processing, 2004.*, pages 541–544, 2004.

[21] Ferenc Bakó, Judit Berkes, and Cecília Szigeti. Households' electricity consumption in hungarian urban areas. *Energies*, 14(10), 2021.

[22] Timothy D Barfoot. *State estimation for robotics*. Cambridge University Press, 2017.

[23] Horace B Barlow. Unsupervised learning. *Neural computation*, 1(3):295–311, 1989.

[24] Mohammadreza Bayat and Antonio P Aguiar. Slam for an auv using vision and an acoustic beacon. *IFAC Proceedings Volumes*, 43(16):503–508, 2010.

[25] Richard Bellman. A markovian decision process. *Journal of mathematics and mechanics*, 6(5):679–684, 1957.

[26] Pierre Benet, Fabien Novella, Marie Ponchart, Pierre Bosser, and Benoit Clement. State-of-the-art of standalone accurate auv positioning - application to high resolution bathymetric surveys. In *OCEANS 2019 - Marseille*, pages 1–10, June 2019.

[27] Yoshua Bengio and Yves Grandvalet. No unbiased estimator of the variance of k-fold cross-validation. *Journal of machine learning research*, 5(Sep):1089–1105, 2004.

[28] Michael R Benjamin, John J Leonard, Henrik Schmidt, and Paul M Newman. An overview of moos-ivp and a brief users guide to the ivp helm autonomy software. *Computer Science and Artificial Intelligence Lab (CSAIL) Technical Reports*.

[29] Viktor Berg. Development and commissioning of a dp system for rov sf 30k. Master's thesis, Institutt for marin teknikk, 2012.

[30] Felix Berkenkamp, Riccardo Moriconi, Angela P Schoellig, and Andreas Krause. Safe learning of regions of attraction for uncertain, nonlinear systems with gaussian processes. In *2016 IEEE 55th Conference on Decision and Control (CDC)*, pages 4661–4666, 2016.

[31] Felix Berkenkamp, Angela P Schoellig, and Andreas Krause. Safe controller optimization for quadrotors with gaussian processes. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 491–496, 2016.

[32] Felix Berkenkamp, Matteo Turchetta, Angela P Schoellig, and Andreas Krause. Safe model-based reinforcement learning with stability guarantees. *arXiv preprint arXiv:1705.08551*, 2017.

[33] Priyadarshi Bhattacharya and Marina L Gavrilova. Voronoi diagram in optimal path planning. In *4th International Symposium on Voronoi Diagrams in Science and Engineering (ISVD 2007)*, pages 38–47, 2007.

[34] Xin-qian Bian, Zheng Qin, and Zhe-ping Yan. Design and evaluation of a hierarchical control architecture for an autonomous underwater vehicle. *Journal of Marine Science and Application*, 7(1):53–58, 2008.

[35] Brian Bingham, Carlos Aguero, Michael McCarrin, Joseph Klamo, Joshua Malia, Kevin Allen, Tyler Lum, Marshall Rawson, and Rumman Waqar. Toward maritime robotic simulation in gazebo. In *Proceedings of MTS/IEEE OCEANS Conference*, Seattle, WA, October 2019.

[36] Christopher M Bishop. *Pattern recognition and machine learning*. Springer, 2006.

[37] Henryk Blasinski, Trisha Lian, and Joyce Farrell. Underwater image systems simulation. In *Imaging and Applied Optics Congress*, 2017.

[38] Ruxandra Bobiti and Mircea Lazar. A sampling approach to finding lyapunov functions for nonlinear discrete-time systems. In *2016 European Control Conference (ECC)*, pages 561–566, 2016.

[39] Vivek S Borkar. A sensitivity formula for risk-sensitive cost and the actor–critic algorithm. *Systems & Control Letters*, 44(5):339–346, 2001.

[40] Olfa Boubaker. The inverted pendulum benchmark in nonlinear control theory: a survey. *International Journal of Advanced Robotic Systems*, 10(5):233, 2013.

[41] Michael Bowling, Johannes Fürnkranz, Thore Graepel, and Ron Musick. Machine learning and games. *Machine learning*, 63(3):211–215, 2006.

[42] Boris Braginsky and Hugo Guterman. Obstacle avoidance approaches for autonomous underwater vehicle: Simulation and experimental results. *IEEE Journal of oceanic engineering*, 41(4):882–892, 2016.

[43] Morten Breivik and Thor I Fossen. Guidance laws for autonomous underwater vehicles. *Underwater vehicles*, 4:51–76, 2009.

[44] J Broussard and Mi O'Brien. Feedforward control to track the output of a forced model. *IEEE Transactions on Automatic control*, 25(4):851–853, 1980.

[45] Paul Burlacu, Vasile Dobref, Nicolae Badara, and Octavian Tarabuta. A lqr controller for an auv depth control. *Annals of DAAAM & Proceedings*, pages 125–127, 2007.

[46] Shehan Caldera, Alexander Rassau, and Douglas Chai. Review of deep learning methods in robotic grasp detection. *Multimodal Technologies and Interaction*, 2(3):57, 2018.

[47] Franck Cameron and Dale E Seborg. A self-tuning controller with a pid structure. In *Real Time Digital Control Application*, pages 613–622. 1984.

[48] Mauro Candeloro, Anastasios M Lekkas, Jeevith Hegde, and Asgeir J Sørensen. A 3d dynamic voronoi diagram-based path-planning system for uuvs. In *OCEANS 2016 MTS/IEEE Monterey*, pages 1–8, 2016.

[49] Ignacio Carlucho, Mariano De Paula, Sebastian A. Villar, and Gerardo G. Acosta. Incremental q-learning strategy for adaptive pid control of mobile robots. *Expert Syst. Appl.*, 80(C):183–199, September 2017.

[50] Ignacio Carlucho, Mariano De Paula, Sen Wang, Bruno V Menna, Yvan R Petillot, and Gerardo G Acosta. Auv position tracking control using end-to-end deep reinforcement learning. In *OCEANS 2018 MTS/IEEE Charleston*, pages 1–8, 2018.

[51] Ignacio Carlucho, Mariano De Paula, Sen Wang, Yvan Petillot, and Gerardo G Acosta. Adaptive low-level control of autonomous underwater vehicles using deep reinforcement learning. *Robotics and Autonomous Systems*, 107:71–86, 2018.

[52] Rich Caruana and Alexandru Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd international conference on Machine learning*, pages 161–168, 2006.

[53] Diogo V Carvalho, Eduardo M Pereira, and Jaime S Cardoso. Machine learning interpretability: A survey on methods and metrics. *Electronics*, 8(8):832, 2019.

[54] Emre Celebi and Kemal Aydin. *Unsupervised learning algorithms*. Springer, 2016.

[55] Nicolò Cesa-Bianchi, Claudio Gentile, Gabor Lugosi, and Gergely Neu. Boltzmann exploration done right. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

[56] Abdelkader Chellabi and Meyer Nahon. Feedback linearization control of undersea vehicles. In *Proceedings of OCEANS'93*, pages I410–I415, 1993.

[57] Huanhuan Chen. *Diversity and regularization in neural network ensembles*. PhD thesis, University of Birmingham, 2008.

[58] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*, pages 7–10, 2016.

[59] Jeffery A Clouse. *On integrating apprentice learning and reinforcement learning*. University of Massachusetts Amherst, 1996.

[60] Jeffery A Clouse and Paul E Utgoff. A teaching method for reinforcement learning. In *Machine learning proceedings 1992*, pages 92–101. 1992.

[61] Karl Cobbe, Oleg Klimov, Chris Hesse, Taehoon Kim, and John Schulman. Quantifying generalization in reinforcement learning. In *International Conference on Machine Learning*, pages 1282–1289, 2019.

[62] Koby Crammer and Yoram Singer. On the algorithmic implementation of multiclass kernel-based vector machines. *Journal of machine learning research*, 2(Dec):265–292, 2001.

[63] Rongxin Cui, Chenguang Yang, Yang Li, and Sanjay Sharma. Adaptive neural network control of auvs with control input nonlinearities using reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 47(6):1019–1029, 2017.

[64] Haskell B Curry. The method of steepest descent for non-linear minimization problems. *Quarterly of Applied Mathematics*, 2(3):258–261, 1944.

[65] João MG Da Silva and Sophie Tarbouriech. Antiwindup design with guaranteed regions of stability: an lmi-based approach. *IEEE Transactions on Automatic Control*, 50(1):106–111, 2005.

[66] Jnaneshwar Das, Julio Harvey, Frédéric Py, Harshvardhan Vathsangam, Rishi Graham, Kanna Rajan, and Gaurav S Sukhatme. Hierarchical probabilistic regression for auv-based adaptive sampling of marine phenomena. In *2013 IEEE International Conference on Robotics and Automation*, pages 5571–5578, 2013.

[67] Maria do Rosário de Pinho, Zahra Foroozandeh, and Aníbal Matos. Optimal control problems for path planing of auv using simplified models. In *2016 IEEE 55th conference on decision and control (CDC)*, pages 210–215, 2016.

[68] Li Deng, Geoffrey Hinton, and Brian Kingsbury. New types of deep neural network learning for speech recognition and related applications: An overview. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 8599–8603, 2013.

[69] Li Deng and Xiao Li. Machine learning paradigms for speech recognition: An overview. *IEEE Transactions on Audio, Speech, and Language Processing*, 21(5):1060–1089, 2013.

[70] Khac D Do and Jie Pan. Global waypoint tracking control of underactuated ships under relaxed assumptions. In *42nd IEEE International Conference on Decision and Control (IEEE Cat. No. 03CH37475)*, volume 2, pages 1244–1249, 2003.

[71] Ürün Dogan, Tobias Glasmachers, and Christian Igel. A unified view on multi-class support vector classification. *J. Mach. Learn. Res.*, 17(45):1–32, 2016.

[72] Huiying Dong, Wenguang Li, Jiayu Zhu, and Shuo Duan. The path planning for mobile robot based on voronoi diagram. In *2010 Third International Conference on Intelligent Networks and Intelligent Systems*, pages 446–449, 2010.

[73] Peter Dorato. A historical review of robust control. *IEEE Control Systems Magazine*, 7(2):44–47, 1987.

[74] Fredrik Dukan. Rov motion control systems. 2014.

[75] Thomas Duriez, Steven L Brunton, and Bernd R Noack. *Machine learning control-taming nonlinear dynamics and turbulence*. Springer, 2017.

[76] Gilberto Echeverria, Nicolas Lassabe, Arnaud Degroote, and Séverin Lemaignan. Modular openrobots simulation engine: Morse. In *Proceedings of the IEEE ICRA*, 2011.

[77] Christopher Edwards and Sarah Spurgeon. *Sliding mode control: theory and applications*. Crc Press, 1998.

[78] Gabriel H Elkaim, Fidelis AP Lie, and Demoz Gebre-Egziabher. Principles of guidance, navigation, and control of uavs. *Handbook of Unmanned Aerial Vehicles*, pages 347–380, 2015.

[79] Taha Elmokadem, Mohamed Zribi, and Kamal Youcef-Toumi. Trajectory tracking sliding mode control of underactuated auvs. *Nonlinear Dynamics*, 84(2):1079–1091, 2016.

[80] Sergei Y Eremenko. Atomic machine learning. *Journal Neurocomputers*, (3), 2018.

[81] Georgios Fagogenis, David Flynn, and David M Lane. Improving underwater vehicle navigation state estimation using locally weighted projection regression. In *2014 IEEE international conference on robotics and automation (ICRA)*, pages 6549–6554, 2014.

[82] William Fedus, Prajit Ramachandran, Rishabh Agarwal, Yoshua Bengio, Hugo Larochelle, Mark Rowland, and Will Dabney. Revisiting fundamentals of experience replay. In *International Conference on Machine Learning*, pages 3061–3071, 2020.

[83] Wladyslaw Findeisen, Frederic N Bailey, Mieczyslaw Brdys, Krzysztof Malinowski, Piotr Tatjewski, and Adam Wozniak. *Control and coordination in hierarchical systems*. John Wiley & Sons, 1980.

[84] Ola E Fjellstad and Thor I Fossen. Singularity-free tracking of unmanned underwater vehicles in 6 dof. In *Proceedings of 1994 33rd IEEE Conference on Decision and Control*, volume 2, pages 1128–1133, 1994.

[85] Thor I Fossen. Guidance and control of ocean vehicles. *University of Trondheim, Norway, Printed by John Wiley & Sons, Chichester, England, ISBN: 0 471 94113 1, Doctors Thesis*, 1999.

[86] Thor I Fossen. *Handbook of marine craft hydrodynamics and motion control*. John Wiley & Sons, 2011.

[87] Gene F Franklin, David J Powell, and Abbas Emami-Naeini. *Feedback control of dynamic systems*. Prentice hall Upper Saddle River, NJ, 2002.

[88] Ding Fu-guang, Jiao Peng, Bian Xin-qian, and Wang Hong-jian. Auv local path planning based on virtual potential field. *IEEE International Conference Mechatronics and Automation, 2005*, 4:1711–1716 Vol. 4, 2005.

[89] Ding Fu-guang, Jiao Peng, Bian Xin-qian, and Wang Hong-Jian. Auv local path planning based on virtual potential field. In *IEEE International Conference Mechatronics and Automation, 2005*, volume 4, pages 1711–1716, 2005.

[90] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*, pages 1587–1596, 2018.

[91] Carlos E Garcia, David M Prett, and Manfred Morari. Model predictive control: Theory and practice—a survey. *Automatica*, 25(3):335–348, 1989.

[92] Javier García and Fernando Fernández. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*, 16(42):1437–1480, 2015.

[93] Luis G García-Valdovinos, Tomás Salgado-Jiménez, Manuel Bandala-Sánchez, Luciano Nava-Balanzar, Rodrigo Hernández-Alvarado, and José A Cruz-Ledesma. Modelling, design and robust control of a remotely operated underwater vehicle. *International Journal of Advanced Robotic Systems*, 11(1):1, 2014.

[94] Murray F Gardner and John L Barnes. *Transients in Linear Systems Studied by the Laplace Transformation*. J. Wiley & Sons, Incorporated, 1942.

[95] Chris Gaskett. Reinforcement learning under circumstances beyond its control. In *Proceedings of the International Conference on Computational Intelligence for Modelling Control and Automation*, 2003.

[96] Chris Gaskett, David Wettergreen, Alexander Zelinsky, et al. Reinforcement learning applied to the control of an autonomous underwater vehicle. In *Proceedings of the Australian conference on robotics and automation (AuCRA99)*, 1999.

[97] Nicolas Gautier, Jean-Luc Aider, Thomas Duriez, Bend R Noack, Marc Segond, and Markus Abel. Closed-loop separation control using machine learning. *Journal of Fluid Mechanics*, 770:442–457, 2015.

[98] Clement Gehring and Doina Precup. Smart exploration in reinforcement learning using absolute temporal difference errors. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, pages 1037–1044, 2013.

[99] Peter Geibel. Reinforcement learning for mdps with constraints. In *European Conference on Machine Learning*, pages 646–653, 2006.

[100] Peter Geibel and Fritz Wysotzki. Risk-sensitive reinforcement learning applied to control under constraints. *Journal of Artificial Intelligence Research*, 24:81–108, 2005.

[101] Alborz Geramifard, Joshua Redding, and Jonathan P How. Intelligent cooperative control architecture: a framework for performance improvement using safe learning. *Journal of Intelligent & Robotic Systems*, 72(1):83–103, 2013.

[102] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019.

[103] Peter Giesl and Sigurdur Hafstein. Review on computational methods for lyapunov functions. *Discrete & Continuous Dynamical Systems-B*, 20(8):2291, 2015.

[104] Bernd Girod, Rudolf Rabenstein, and Alexander Stenger. *Signals and Systems*. Wiley, 2001.

[105] Paul Glasserman. *Monte Carlo methods in financial engineering*. Springer Science & Business Media, 2013.

[106] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[107] Julian Gonzalez, Andreina Benezra, Spartacus Gomariz, and David Sarriá. Limitations of linear control for cormoran-auv. In *2012 IEEE International Instrumentation and Measurement Technology Conference Proceedings*, pages 1726–1729, 2012.

[108] Louis A Gonzalez. Design, modelling and control of an autonomous underwater vehicle. *BE Thesis, The University of Western Australia, Australia*, 2004.

[109] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[110] Neil J Gordon, David J Salmond, and Adrian FM Smith. Novel approach to nonlinear/non-gaussian bayesian state estimation. In *IEE proceedings F (radar and signal processing)*, volume 140, pages 107–113, 1993.

[111] Abhijit Gosavi. Reinforcement learning for model building and variance-penalized control. In *Proceedings of the 2009 Winter Simulation Conference (WSC)*, pages 373–379, 2009.

[112] Joshua G Graver and Naomi E Leonard. Underwater glider dynamics and control. In *12th international symposium on unmanned untethered submersible technology*, pages 1742–1710, 2001.

[113] Ørjan Grefstad and Ingrid Schjølberg. Navigation and collision avoidance of underwater vehicles using sonar data. In *2018 IEEE/OES Autonomous Underwater Vehicle Workshop (AUV)*, pages 1–6, 2018.

[114] Jen-Hwa Guo, Fong-Chen Chiu, and C-C Huang. Design of a sliding mode fuzzy controller for the guidance and control of an autonomous underwater vehicle. *Ocean Engineering*, 30(16):2137–2155, 2003.

[115] Tuomas Haarnoja, Haoran Tang, Pieter Abbeel, and Sergey Levine. Reinforcement learning with deep energy-based policies. In *International Conference on Machine Learning*, pages 1352–1361, 2017.

[116] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, pages 1861–1870, 2018.

[117] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, et al. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018.

[118] Roland Hafner and Martin Riedmiller. Reinforcement learning in feedback control. *Machine learning*, 84(1-2):137–169, 2011.

[119] Mohanad M Hammad, Ahmed K Elshenawy, and MI El Singaby. Position control and stabilization of fully actuated auv using pid controller. In *Proceedings of SAI Intelligent Systems Conference*, pages 517–536, 2016.

[120] Lars P Hansen and Thomas J Sargent. Robust control and model uncertainty. *American Economic Review*, 91(2):60–66, 2001.

[121] Mohammad Hassanzadeh and Cansin Y Evrenosoglu. A regression analysis based state transition model for power system dynamic state estimation. In *2011 North American Power Symposium*, pages 1–5, 2011.

[122] Hado Hasselt. Double q-learning. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc., 2010.

[123] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. Overview of supervised learning. In *The elements of statistical learning*, pages 9–41. 2009.

[124] Trevor Hastie, Robert Tibshirani, and Jerome H Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2009.

[125] Simen T Havenstrøm, Adil Rasheed, and Omer San. Deep reinforcement learning controller for 3d path following and collision avoidance by autonomous underwater vehicles. *Frontiers in Robotics and AI*, 7:211, 2021.

[126] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[127] Pingan He and Sarangapani Jagannathan. Reinforcement learning neural-network-based controller for nonlinear discrete-time systems with input constraints. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 37(2):425–436, 2007.

[128] Anthony J Healey and David Lienard. Multivariable sliding mode control for autonomous diving and steering of unmanned underwater vehicles. *IEEE journal of Oceanic Engineering*, 18(3):327–339, 1993.

[129] Matthias Heger. Consideration of risk in reinforcement learning. In *Machine Learning Proceedings 1994*, pages 105–111. 1994.

[130] Matthias Heger. *Risk and reinforcement learning: concepts and dynamic programming*. ZKW, 1994.

[131] Robert Hermann and Arthur Krener. Nonlinear controllability and observability. *IEEE Transactions on automatic control*, 22(5):728–740, 1977.

[132] Alex Hernández-García and Peter König. Data augmentation instead of explicit regularization. *arXiv preprint arXiv:1806.03852*, 2018.

[133] Rodrigo Hernández-Alvarado, Luis G García-Valdovinos, Tomás Salgado-Jiménez, Alfonso Gómez-Espinosa, and Fernando Fonseca-Navarro. Neural network-based self-tuning pid control for underwater vehicles. *Sensors*, 16(9), 2016.

[134] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *Cited on*, 14(8), 2012.

[135] Douglas P Horner, Anthony J Healey, and Sean P Kragelund. Auv experiments in obstacle avoidance. In *Proceedings of OCEANS 2005 MTS/IEEE*, pages 1464–1470, 2005.

[136] Masaru Hoshiya and Etsuro Saito. Structural identification by extended kalman filter. *Journal of engineering mechanics*, 110(12):1757–1770, 1984.

[137] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. *Applied logistic regression*. John Wiley & Sons, 2013.

[138] Ronald A Howard. Dynamic programming and markov processes. 1960.

[139] Ronald A Howard and James E Matheson. Risk-sensitive markov decision processes. *Management science*, 18(7):356–369, 1972.

[140] Mark N Howell and Matt C Best. On-line pid tuning for engine idle-speed control using continuous action reinforcement learning automata. *Control Engineering Practice*, 8(2):147–154, 2000.

[141] Bo Hu, Hai Tian, Jiani Qian, Guochao Xie, Linlang Mo, and Shuo Zhang. A fuzzy-pid method to improve the depth control of auv. In *2013 IEEE International Conference on Mechatronics and Automation*, pages 1528–1533, 2013.

[142] Peter J Huber. Robust estimation of a location parameter. In *Breakthroughs in statistics*, pages 492–518. 1992.

[143] John D Hunter. Matplotlib: A 2d graphics environment. *IEEE Annals of the History of Computing*, 9(03):90–95, 2007.

[144] Yujia Huo, Yiping Li, and Xisheng Feng. Model-free recurrent reinforcement learning for auv horizontal control. In *IOP Conference Series: Materials Science and Engineering*, volume 428, page 012063, 2018.

[145] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the 5th international conference on Learning and Intelligent Optimization*, pages 507–523, 2011.

[146] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, editors. *Automated Machine Learning: Methods, Systems, Challenges*. Springer, 2018.

[147] Marcus Hutter. Self-optimizing and pareto-optimal policies in general environments based on bayes-mixtures. In *International Conference on Computational Learning Theory*, pages 364–379, 2002.

[148] Jimin Hwang, Shuangshuang Fan, Peter King, and Alexander Forrest. Development of error reduction model using bayesian filter for auv navigating under moving ice. In *2018 IEEE/OES Autonomous Underwater Vehicle Workshop (AUV)*, pages 1–6, 2018.

[149] Tadahiro Hyakudome. Design of autonomous underwater vehicle. *International Journal of Advanced Robotic Systems*, 8(1):9, 2011.

[150] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456, 2015.

[151] Alberto Isidori. *Nonlinear control systems*. Springer Science & Business Media, 2013.

[152] Haider K Jabbar and Rafiqul Z Khan. Methods to avoid over-fitting and under-fitting in supervised machine learning (comparative study). *Computer Science, Communication and Instrumentation Devices*, pages 163–172, 2015.

[153] Max Jaderberg, Volodymyr Mnih, Wojciech M Czarnecki, Tom Schaul, Joel Z Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. *arXiv preprint arXiv:1611.05397*, 2016.

[154] Gareth James, Daniela Witten, Trevor Hastie, and Rob Tibshirani. *An Introduction to Statistical Learning: with Applications in R*. Springer, 2014.

[155] Sooyoung Jang and Hyung-Il Kim. Entropy-aware model initialization for effective exploration in deep reinforcement learning. *arXiv preprint arXiv:2108.10533*, 2021.

[156] Katarzyna Janocha and Wojciech M Czarnecki. On loss functions for deep neural networks in classification. *arXiv preprint arXiv:1702.05659*, 2017.

[157] Luc Jaulin. *Automation for robotics*. John Wiley & Sons, 2015.

[158] Luc Jaulin. *Mobile robotics*. John Wiley & Sons, 2019.

[159] Luc Jaulin, Michel Kieffer, Olivier Didrit, and Eric Walter. Interval analysis. In *Applied interval analysis*, pages 11–43. 2001.

[160] Luc Jaulin and Fabrice Le Bars. A simple controller for line following of sailboats. In *Robotic Sailing 2012*, pages 117–129. 2013.

[161] Scott A Jenkins, Douglas E Humphreys, Jeff Sherman, Jim Osse, Clayton Jones, Naomi E Leonard, Joshua Graver, Ralf Bachmayer, Ted Clem, Paul Carroll, et al. Underwater glider system study. 2003.

[162] Michael A Johnson and Mohammad H Moradi. *PID control*. Springer, 2005.

[163] Simon J Julier and Jeffrey K Uhlmann. Unscented filtering and nonlinear estimation. *Proceedings of the IEEE*, 92(3):401–422, 2004.

[164] Yoshinobu Kadota, Masami Kurano, and Masami Yasuda. Discounted markov decision processes with utility constraints. *Computers & Mathematics with Applications*, 51(2):279–284, 2006.

[165] Rickard Karlsson and Fredrik Gustafsson. Bayesian surface and underwater navigation. *IEEE Transactions on Signal Processing*, 54(11):4204–4213, 2006.

[166] Artúr I Károly, Péter Galambos, József Kuti, and Imre J Rudas. Deep learning in robotics: Survey on model structures and training strategies. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 51(1):266–279, 2020.

[167] Anatole Katok and Boris Hasselblatt. *Introduction to the modern theory of dynamical systems*. Cambridge university press, 1997.

[168] Motohiro Kawafuku, Minoru Sasaki, and Shinya Kato. Self-tuning pid control of a flexible micro-actuator using neural networks. In *SMC'98 Conference Proceedings. 1998 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No. 98CH36218)*, volume 3, pages 3067–3072, 1998.

[169] Konstantin G Kebkal and Andrei I Mashoshin. Auv acoustic positioning methods. *Gyroscopy and navigation*, 8(1):80–89, 2017.

[170] Marzuki Khalid and Sigeru Omatu. A neural network controller for a temperature control system. *IEEE control systems magazine*, 12(3):58–64, 1992.

[171] Hassan K Khalil. *Nonlinear Systems*. Prentice Hall, Upper Saddle River, NJ, 2002.

[172] Oussama Khatib. Real-time obstacle avoidance for manipulators and mobile robots. In *Autonomous robot vehicles*, pages 396–404. 1986.

[173] Mohammad H Khodayari and Saeed Balochian. Modeling and control of autonomous underwater vehicle (auv) in heading and depth attitude via self-adaptive fuzzy pid controller. *Journal of Marine Science and Technology*, 20(3):559–578, 2015.

[174] Michel Kieffer, Luc Jaulin, and Eric Walter. Guaranteed recursive nonlinear state estimation using interval analysis. In *Proceedings of the 37th IEEE Conference on Decision and Control (Cat. No. 98CH36171)*, volume 4, pages 3966–3971, 1998.

[175] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[176] Stephen C Kleene. Representations of events in nerve nets and finite automata. *Automata Studies [Annals of Math. Studies 34]*, 1956.

[177] David G Kleinbaum and Mitchell Klein. *Logistic regression*. Springer, 2010.

[178] Stefan Knerr, Léon Personnaz, and Gérard Dreyfus. Handwritten digit recognition by neural networks with single-layer training. *IEEE Transactions on neural networks*, 3(6):962–968, 1992.

[179] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[180] William Koch, Renato Mancuso, Richard West, and Azer Bestavros. Reinforcement learning for uav attitude control. *ACM Transactions on Cyber-Physical Systems*, 3(2):1–21, 2019.

[181] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2149–2154, 2004.

[182] HN Koivo and JT Tanttu. Tuning of pid conrollers: survey of siso and mimo techniques. In *Intelligent tuning and adaptive control*, pages 75–80. 1991.

[183] Krzysztof Kowalski and Willi-hans Steeb. *Nonlinear dynamical systems and Carleman linearization*. World Scientific, 1991.

[184] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[185] Gregory Kuhlmann, Peter Stone, Raymond Mooney, and Jude Shavlik. Guiding a reinforcement learner with natural language advice: Initial results in robocup soccer. In *The AAAI-2004 workshop on supervisory control of learning and adaptive systems*, 2004.

[186] Praveen Kulkarni, Joaquin Zepeda, Frederic Jurie, Patrick Pérez, and Louis Chevallier. Learning the structure of deep architectures using l1 regularization. In *British Machine Vision Conference, 2015*, 2015.

[187] Solomon Kullback. Letter to the editor: The kullback–leibler distance. *The American Statistician*, 41(4):340–341, 1987.

[188] Solomon Kullback. *Information theory and statistics*. Courier Corporation, 1997.

[189] Akhilesh Kumar, Ratna B Chinnam, and Finn Tseng. An hmm and polynomial regression based approach for remaining useful life and health state estimation of cutting tools. *Computers & Industrial Engineering*, 128:1008–1014, 2019.

[190] Harold J Kushner. Stochastic stability and control. Technical report, Brown Univ Providence RI, 1967.

[191] Joseph P La Salle. *The stability of dynamical systems*. SIAM, 1976.

[192] CE Langenhop. On the stabilization of linear systems. *Proceedings of the American Mathematical Society*, 15(5):735–742, 1964.

[193] Lionel Lapierre and Bruno Jouvencel. Robust nonlinear path-following control of an auv. *IEEE Journal of Oceanic Engineering*, 33(2):89–102, 2008.

[194] Martin Larsen. Industrial applications of fuzzy logic control. *International Journal of Man-Machine Studies*, 12(1):3–10, 1980.

[195] Jean-Claude Latombe. *Robot motion planning*. Springer Science & Business Media, 2012.

[196] Steven M LaValle. *Planning algorithms*. Cambridge university press, 2006.

[197] Edith LM Law, Melanie Coggan, Doina Precup, and Bohdana Ratitch. Risk-directed exploration in reinforcement learning. *Planning and Learning in A Priori Unknown or Dynamic Domains*, 97, 2005.

[198] Quoc V Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Kai Chen, Greg S Corrado, Jeff Dean, and Andrew Y Ng. Building high-level features using large scale unsupervised learning. In *Proceedings of the 29th International Coference on International Conference on Machine Learning*, ICML'12, page 507–514, Madison, WI, USA, 2012. Omnipress.

[199] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[200] Yann LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. 2012.

[201] Yann LeCun, Lawrence D Jackel, Leon Bottou, A Brunot, Corinna Cortes, John S Denker, Harris Drucker, Isabelle Guyon, UA Muller, Eduard Sackinger, et al. Comparison of learning algorithms for handwritten digit recognition. In *International conference on artificial neural networks*, volume 60, pages 53–60, 1995.

[202] Chengming Lee and Rongshun Chen. Optimal self-tuning pid controller based on low power consumption for a server fan cooling system. *Sensors*, 15(5):11685–11700, 2015.

[203] Ching-Hung Lee. A survey of pid controller design based on gain and phase margins. *International Journal of Computational Cognition*, 2(3):63–100, 2004.

[204] Daewon Lee, Jin H Kim, and Shankar Sastry. Feedback linearization vs. adaptive sliding mode control for a quadrotor helicopter. *International Journal of control, Automation and systems*, 7(3):419–428, 2009.

[205] Alexandre Lefort, Xavier Dal Santo, Jordan Ninin, and Benoit Clement. Depth control of a submarine: An application of structured H_∞ synthesis method for uncertain models based on interval analysis. In *2018 Australian & New Zealand Control Conference (ANZCC)*, pages 269–274, 2018.

[206] Erich L Lehmann and George Casella. *Theory of point estimation*. Springer Science & Business Media, 2006.

[207] Claude Lemaréchal. Cauchy and the gradient method. *Doc Math Extra*, 251(254):10, 2012.

[208] John J Leonard and Alexander Bahr. Autonomous underwater vehicle navigation. *Springer handbook of ocean engineering*, pages 341–358, 2016.

[209] Frank L Lewis, Draguna Vrabie, and Vassilis L Syrmos. *Optimal control*. John Wiley & Sons, 2012.

[210] Ye Li, Yan-qing Jiang, Lei-feng Wang, Jian Cao, and Guo-cheng Zhang. Intelligent pid guidance control for auv path tracking. *Journal of Central South University*, 22(9):3440–3449, 2015.

[211] Xiao Liang, Lei Wan, James IR Blake, Ajit R Shenoi, and Nicholas Townsend. Path following of an underactuated auv based on fuzzy backstepping sliding mode control. *International Journal of Advanced Robotic Systems*, 13(3):122, 2016.

[212] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas MO Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015.

[213] Shifei Liu, Yanhui Wei, and Yanbin Gao. 3d path planning for auv using fuzzy logic. In *2012 International conference on computer science and information processing (CSIP)*, pages 599–603, 2012.

[214] Shuyong Liu, Danwei Wang, and Engkee Poh. Output feedback control design for station keeping of auvs under shallow water wave disturbances. *International Journal of Robust and Nonlinear Control: IFAC-Affiliated Journal*, 19(13):1447–1470, 2009.

[215] Cedric L Logan. A comparison between h-infinity/mu-synthesis control and sliding-mode control for robust control of a small autonomous underwater vehicle. In *Proceedings of IEEE Symposium on Autonomous Underwater Vehicle Technology (AUV'94)*, pages 399–416, 1994.

[216] Jørgen Lorentz and Junku Yuh. A survey and experimental study of neural network auv control. In *Proceedings of Symposium on Autonomous Underwater Vehicle Technology*, pages 109–116, 1996.

[217] David G Luenberger et al. Investment science. *OUP Catalogue*, 1997.

[218] Juan CC Luque and Décio C Donha. Auv identification and robust control. *IFAC Proceedings Volumes*, 44(1):14735–14741, 2011.

[219] Mark Lutz. *Learning python: Powerful object-oriented programming.* " O'Reilly Media, Inc.", 2013.

[220] Aleksandr M Lyapunov. The general problem of the stability of motion. *International journal of control*, 55(3):531–534, 1992.

[221] Frederic Maire and Vadim Bulitko. Apprenticeship learning for initial value functions in reinforcement learning. *Planning and Learning in A Priori Unknown or Dynamic Domains*, page 23, 2005.

[222] Musa MM Manhães, Sebastian A Scherer, Martin Voss, Luiz R Douat, and Thomas Rauschenbach. Uuv simulator: A gazebo-based package for underwater intervention and multi-robot simulation. In *OCEANS 2016 MTS/IEEE Monterey*, pages 1–8, 2016.

[223] Nadav D Marom, Lior Rokach, and Armin Shmilovici. Using the confusion matrix for improving ensemble classifiers. In *2010 IEEE 26-th Convention of Electrical and Electronics Engineers in Israel*, pages 000555–000559, 2010.

[224] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[225] Conor McGann, Frederic Py, Kanna Rajan, John P Ryan, and Richard Henthorn. Adaptive control for autonomous underwater vehicles. In *AAAI*, pages 1319–1324, 2008.

[226] Oliver Mihatsch and Ralph Neuneier. Risk-sensitive reinforcement learning. *Machine learning*, 49(2-3):267–290, 2002.

[227] Paul A Miller, Jay A Farrell, Yuanyuan Zhao, and Vladimir Djapic. Autonomous underwater vehicle navigation. *IEEE Journal of Oceanic Engineering*, 35(3):663–678, 2010.

[228] Tom M Mitchell. Machine learning and data mining. *Communications of the ACM*, 42(11):30–36, 1999.

[229] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[230] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[231] Signe Moe, Kristin Y Pettersen, Thor I Fossen, and Jan T Gravdahl. Line-of-sight curved path following for underactuated usvs and auvs in the horizontal plane under the influence of ocean currents. *2016 24th Mediterranean Conference on Control and Automation (MED)*, pages 38–45, 2016.

[232] Teodor M Moldovan and Pieter Abbeel. Safe exploration in markov decision processes. In *Proceedings of the 29th International Coference on International Conference on Machine Learning*, pages 1451–1458, 2012.

[233] Cleve Moler. Design of an interactive matrix calculator. In *Proceedings of the May 19-22, 1980, national computer conference*, pages 363–368, 1980.

[234] John Moody and Christian J Darken. Fast learning in networks of locally-tuned processing units. *Neural computation*, 1(2):281–294, 1989.

[235] Steven W Moore, Harry Bohm, Vickie Jensen, and Nola Johnston. *Underwater robotics: science, design & fabrication*. MATE, 2010.

[236] Manfred Morari and Jay H Lee. Model predictive control: past, present and future. *Computers & Chemical Engineering*, 23(4-5):667–682, 1999.

[237] Fernando Morilla, Francisco Vázquez, and R Hernández. Pid control design with guaranteed stability. *IFAC Proceedings Volumes*, 39(6):13–18, 2006.

[238] Tetsuro Morimura, Masashi Sugiyama, Hisashi Kashima, Hirotaka Hachiya, and Toshiyuki Tanaka. Nonparametric return distribution approximation for reinforcement learning. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, pages 799–806, 2010.

[239] Tetsuro Morimura, Masashi Sugiyama, Hisashi Kashima, Hirotaka Hachiya, and Toshiyuki Tanaka. Parametric return density estimation for reinforcement learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, pages 368–375, 2010.

[240] Amir Mosavi and Annamaria Varkonyi. Learning in robotics. *International Journal of Computer Applications*, 157(1):8–11, 2017.

[241] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[242] Anirban Nag, Surendra S Patel, Kaushal Kishore, and Shaikh A Akbar. A robust h-infinity based depth control of an autonomous underwater vehicle. In *2013 International Conference on Advanced Electronic Systems (ICAES)*, pages 68–73, 2013.

[243] Kento Nakai and Kenji Uchiyama. Vector fields for uav guidance using potential function method for formation flight. In *AIAA Guidance, Navigation, and Control (GNC) Conference*, page 4626, 2013.

[244] Salih N Neftci. *Principles of financial engineering*. Academic Press, 2008.

[245] Anil Nerode and Wolf Kohn. An autonomous systems control theory: An overview. In *IEEE Symposium on Computer-Aided Control System Design*, pages 204–210, 1992.

[246] Renata Neuland, Renan Maffei, Luc Jaulin, Edson Prestes, and Mariana Kolberg. Improving the precision of auvs localization in a hybrid interval-probabilistic approach using a set-inversion strategy. *Unmanned Systems*, 2(04):361–375, 2014.

[247] Arnab Nilim and Laurent El Ghaoui. Robust control of markov decision processes with uncertain transition matrices. *Operations Research*, 53(5):780–798, 2005.

[248] Jose R Noriega and Hong Wang. A direct adaptive neural-network control for unknown nonlinear systems and its application. *IEEE transactions on neural networks*, 9(1):27–34, 1998.

[249] Vilém Novák, Irina Perfilieva, and Jiri Mockor. *Mathematical principles of fuzzy logic*. Springer Science & Business Media, 2012.

[250] Ismoilov Nusrat and Sung-Bong Jang. A comparison of regularization techniques in deep neural networks. *Symmetry*, 10(11):648, 2018.

[251] Myung-Hwan Oh, Jun-Ho Oh, et al. Homing and docking control of auv using model predictive control. In *The Fifth ISOPE Pacific/Asia Offshore Mechanics Symposium*, 2002.

[252] Travis E Oliphant. *A guide to NumPy*. Trelgol Publishing USA, 2006.

[253] Linlin Ou, Youchun Tang, Danying Gu, and Weidong Zhang. Stability analysis of pid controllers for integral processes with time delay. In *Proceedings of the 2005, American Control Conference, 2005.*, pages 4247–4252, 2005.

[254] Li-xin Pan, Hong-zhang Jin, and Lin-lin Wang. Robust control based on feedback linearization for roll stabilizing of autonomous underwater vehicle under wave disturbances. *China Ocean Engineering*, 25(2):251, 2011.

[255] Marcelo Paravisi, Davi H Santos, Vitor Jorge, Guilherme Heck, Luiz M Gonçalves, and Alexandre Amory. Unmanned surface vehicle simulator with realistic environmental disturbances. *Sensors*, 19(5), 2019.

[256] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*, 2019.

[257] Mihir Patil, Bilal Wehbe, and Matias Valdenegro-Toro. Deep reinforcement learning for continuous docking control of autonomous underwater vehicles: A benchmarking study. *arXiv preprint arXiv:2108.02665*, 2021.

[258] Balasaheb M Patre, Pandurang S Londhe, Laxman M Waghmare, and Mohan Santhakumar. Disturbance estimator based non-singular fast fuzzy terminal sliding mode control of an autonomous underwater vehicle. *Ocean Engineering*, 159:372–387, 2018.

[259] Sachin C Patwardhan, Shankar Narasimhan, Prakash Jagadeesan, Bhushan Gopaluni, and Sirish L Shah. Nonlinear bayesian state estimation: A review of recent developments. *Control Engineering Practice*, 20(10):933–953, 2012.

[260] Liam Paull, Sajad Saeedi, Mae Seto, and Howard Li. Auv navigation and localization: A review. *IEEE Journal of oceanic engineering*, 39(1):131–149, 2013.

[261] Wilfrid Perruquetti and Jean-Pierre Barbot. *Sliding mode control in engineering*. CRC press, 2002.

[262] Jan Peters and Stefan Schaal. Policy gradient methods for robotics. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2219–2225, 2006.

[263] Jan Petrich. *Improved guidance, navigation, and control for autonomous underwater vehicles: theory and experiment*. PhD thesis, Virginia Tech, 2009.

[264] Huy X Pham, Hung M La, David Feil-Seifer, and Luan V Nguyen. Autonomous uav navigation using reinforcement learning. *arXiv preprint arXiv:1801.05086*, 2018.

[265] Harry A Pierson and Michael S Gashler. Deep learning in robotics: a review of recent research. *Advanced Robotics*, 31(16):821–835, 2017.

[266] Mario Prats, Javier Perez, Javier J Fernandez, and Pedro J Sanz. An open source tool for simulation and supervision of underwater intervention missions. In *2012 IEEE/RSJ international conference on Intelligent Robots and Systems*, pages 2577–2582, 2012.

[267] Lutz Prechelt. Early stopping-but when? In *Neural Networks: Tricks of the trade*, pages 55–69. 1998.

[268] Demetri Psaltis, Athanasios Sideris, and Alan A Yamamura. A multilayered neural network controller. *IEEE control systems magazine*, 8(2):17–21, 1988.

[269] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5, 2009.

[270] Sebastian Raschka and Vahid Mirjalili. *Python Machine Learning, 2nd Ed.* Packt Publishing, 2017.

[271] KRG Reshmi and PS Priya. Design and control of autonomous unerwater vehicle for depth control using lqr controller. *International Journal of Science and Research (IJSR)*, 5(7), 2016.

[272] Spencer M Richards, Felix Berkenkamp, and Andreas Krause. The lyapunov neural network: Adaptive stability certification for safe learning of dynamical systems. *arXiv preprint arXiv:1808.00924*, 2018.

[273] Pere Ridao, Junku Yuh, Joan Batlle, and Kazuo Sugihara. On auv control architecture. In *Proceedings. 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000)(Cat. No. 00CH37113)*, volume 2, pages 855–860, 2000.

[274] Emilie Roche, Olivier Sename, Daniel Simon, and Sébastien Varrier. A hierarchical varying sampling h control of an auv. *IFAC Proceedings Volumes*, 44(1):14729–14734, 2011.

[275] Eric Rohmer, Surya PN Singh, and Marc Freese. V-rep: A versatile and scalable robot simulation framework. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1321–1326, 2013.

[276] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[277] Michael T Rosenstein and Andrew G Barto. Supervised actor-critic reinforcement learning. *Handbook of Learning and Approximate Dynamic Programming*, pages 359–380, 2004.

[278] Spandan Roy, Sankar N Shome, Subhajit Nandy, Ranjit Ray, and Virendra Kumar. Trajectory following control of auv: a robust approach. *Journal of The Institution of Engineers (India): Series C*, 94(3):253–265, 2013.

[279] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[280] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[281] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*. University of Cambridge, 1994.

[282] Avilash Sahoo, Santosha K Dwivedy, and PS Robi. Advancements in the field of autonomous underwater vehicle. *Ocean Engineering*, 181:145–160, 2019.

[283] Tomás Salgado-Jiménez, Luis G García-Valdovinos, Guillermo Delgado-Ramírez, and Andrzej Bartoszewicz. Control of rovs using a model-free 2nd-order sliding mode approach. *Sliding mode control*, pages 347–368, 2011.

[284] Tomas Salgado-Jimenez, Jean-Mathias Spiewak, Philippe Fraisse, and Bruno Jouvencel. A robust control algorithm for auv: Based on a high order sliding mode. In *Oceans' 04 MTS/IEEE Techno-Ocean'04 (IEEE Cat. No. 04CH37600)*, volume 1, pages 276–281, 2004.

[285] Arthur L Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229, 1959.

[286] Timothy Sands and Kevin Bollino. Autonomous underwater vehicle guidance, navigation, and control. In *Autonomous Vehicles*. 2018.

[287] Mohan Santhakumar and Thondiyath Asokan. Coupled, non-linear control system design for autonomous underwater vehicle (auv). In *2008 10th International Conference on Control, Automation, Robotics and Vision*, pages 2309–2313, 2008.

[288] Mohan Santhakumar and Thondiyath Asokan. Power efficient dynamic station keeping control of a flat-fish type autonomous underwater vehicle through design modifications of thruster configuration. *Ocean Engineering*, 58:11–21, 2013.

[289] Makoto Sato, Hajime Kimura, and Shibenobu Kobayashi. Td algorithm for the variance of return and mean-variance reinforcement learning. *Transactions of the Japanese Society for Artificial Intelligence*, 16(3):353–362, 2001.

[290] Stefan Schaal et al. Learning from demonstration. *Advances in neural information processing systems*, pages 1040–1046, 1997.

[291] Cullen Schaffer. Selecting a classification method by cross-validation. *Machine Learning*, 13(1):135–143, 1993.

[292] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[293] Miloš Schlegel and Jiří Mertl. Stability regions for pi/pid controller and matlab program. In *5th international carpathian control conference*, 2004.

[294] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897, 2015.

[295] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[296] Nicu Sebe, Ira Cohen, Ashutosh Garg, and Thomas S Huang. *Machine learning in computer vision*. Springer Science & Business Media, 2005.

[297] Konstantinos Sechidis, Grigorios Tsoumakas, and Ioannis Vlahavas. On the stratification of multi-label data. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 145–158, 2011.

[298] Mohamed SI Seddik, Luc Jaulin, and Jonathan Grimsdale. Phase based localization for underwater vehicles using interval analysis. *Mathematics in Computer Science*, 8(3):495–502, 2014.

[299] Mostafa Sedighizadeh and Alireza Rezazadeh. Adaptive pid controller based on reinforcement learning for wind turbine control. In *Proceedings of world academy of science, engineering and technology*, volume 27, pages 257–262, 2008.

[300] Kun Shao, Zhentao Tang, Yuanheng Zhu, Nannan Li, and Dongbin Zhao. A survey of deep reinforcement learning in video games. *arXiv preprint arXiv:1912.10944*, 2019.

[301] Xinhui Shao, Bo He, Jia Guo, and Tianhong Yan. The application of auv navigation based on adaptive extended kalman filter. In *Oceans 2016-Shanghai*, pages 1–4, 2016.

[302] Chao Shen, Yang Shi, and Brad Buckham. Trajectory tracking control of an autonomous underwater vehicle using lyapunov-based model predictive control. *IEEE Transactions on Industrial Electronics*, 65(7):5796–5805, 2017.

[303] Chao Shen, Yang Shi, and Brad Buckham. Path-following control of an auv: A multiobjective model predictive control approach. *IEEE Transactions on Control Systems Technology*, 27(3):1334–1342, 2018.

[304] Huailin Shu and Youguo Pi. Pid neural networks for time-delay systems. *Computers & Chemical Engineering*, 24(2-7):859–862, 2000.

[305] Bruno Siciliano and Oussama Khatib. *Springer handbook of robotics*. springer, 2016.

[306] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *International conference on machine learning*, pages 387–395, 2014.

[307] Jean-Jacques E Slotine, Weiping Li, et al. *Applied nonlinear control*. Prentice hall Englewood Cliffs, NJ, 1991.

[308] Samuel M Smith, Graeme JS Rae, Taylor D Anderson, and Andy M Shein. Fuzzy logic control of an autonomous underwater vehicle. *Control Engineering Practice*, 2(2):321–331, 1994.

[309] Yoann Sola, Thomas Chaffre, Gilles Le Chenadec, Karl Sammut, and Benoit Clement. Evaluation of a deep-reinforcement-learning-based controller for the control of an autonomous underwater vehicle. In *Global Oceans 2020: Singapore–US Gulf Coast*, pages 1–7, 2020.

[310] Yong Song, Yi-bin Li, Cai-hong Li, and Gui-fang Zhang. An efficient initialization approach of q-learning for mobile robots. *International Journal of Control, Automation and Systems*, 10(1):166–172, 2012.

[311] Christian R Sonnenburg and Craig A Woolsey. Modeling, identification, and control of an unmanned surface vehicle. *Journal of Field Robotics*, 30(3):371–398, 2013.

[312] Polihronis-Thomas D Spanos. Linearization techniques for non-linear dynamical systems. 1976.

[313] Mark W Spong. Partial feedback linearization of underactuated mechanical systems. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'94)*, volume 1, pages 314–321, 1994.

[314] Ingo Steinwart. How to compare different loss functions and their risks. *Constructive Approximation*, 26(2):225–287, 2007.

[315] Roland Stelzer and Tobias Pröll. Autonomous sailboat navigation for short course racing. *Robotics and autonomous systems*, 56(7):604–614, 2008.

[316] Yushan Sun, Junhan Cheng, Guocheng Zhang, and Hao Xu. Mapless motion planning system for an autonomous underwater vehicle using policy gradient-based deep reinforcement learning. *Journal of Intelligent & Robotic Systems*, 96(3):591–601, 2019.

[317] Niko Sünderhauf, Oliver Brock, Walter Scheirer, Raia Hadsell, Dieter Fox, Jürgen Leitner, Ben Upcroft, Pieter Abbeel, Wolfram Burgard, Michael Milford, et al. The limits and potentials of deep learning for robotics. *The International Journal of Robotics Research*, 37(4-5):405–420, 2018.

[318] Su W Sung and In-Beum Lee. Limitations and countermeasures of pid controllers. *Industrial & Engineering Chemistry Research*, 35(8):2596–2610, 1996.

[319] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

[320] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[321] Richard S Sutton, David A McAllester, Satinder P Singh, Yishay Mansour, et al. Policy gradient methods for reinforcement learning with function approximation. In *NIPs*, volume 99, pages 1057–1063, 1999.

[322] Min-Jea Tahk, Chang-Su Park, and Chang-Kyung Ryoo. Line-of-sight guidance laws for formation flight. *Journal of Guidance, Control, and Dynamics*, 28(4):708–716, 2005.

[323] Aviv Tamar, Dotan Di Castro, and Shie Mannor. Policy gradients with variance related risk criteria. In *Proceedings of the 29th International Coference on International Conference on Machine Learning*, pages 1651–1658, 2012.

[324] Aviv Tamar, Huan Xu, and Shie Mannor. Scaling up robust mdps by reinforcement learning. *arXiv preprint arXiv:1306.6189*, 2013.

[325] Mingxing Tan and Quoc V Le. Efficientnet: Rethinking model scaling for convolutional neural networks, 2020.

[326] Jie Tang, Arjun Singh, Nimbus Goehausen, and Pieter Abbeel. Parameterized maneuver learning for autonomous helicopter flight. In *2010 IEEE International Conference on Robotics and Automation*, pages 1142–1148, 2010.

[327] Gang Tao. *Adaptive control design and analysis*. John Wiley & Sons, 2003.

[328] Adi L Tarca, Vincent J Carey, Xue-wen Chen, Roberto Romero, and Sorin Drăghici. Machine learning and its applications to biology. *PLoS Comput Biol*, 3(6):e116, 2007.

[329] Matthew E Taylor and Peter Stone. Representation transfer for reinforcement learning. In *AAAI Fall Symposium: Computational Approaches to Representation Change during Learning and Development*, pages 78–85, 2007.

[330] Matthew E Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(7), 2009.

[331] Andrea L Thomaz and Cynthia Breazeal. Teachable robots: Understanding human teaching behavior to build more effective robot learners. *Artificial Intelligence*, 172(6-7):716–737, 2008.

[332] William R Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3–4):285–294, 1933.

[333] Dugald Thomson and Stan Dosso. *AUV localization in an underwater acoustic positioning system*. IEEE, 2013.

[334] Ruben R Torrado, Philip Bontrager, Julian Togelius, Jialin Liu, and Diego Perez-Liebana. Deep reinforcement learning for general video game ai. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, 2018.

[335] Thomas Tosik and Erik Maehle. Mars: A simulation environment for marine robotics. In *2014 Oceans-St. John's*, pages 1–7, 2014.

[336] John N Tsitsiklis and Benjamin Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE transactions on automatic control*, 42(5):674–690, 1997.

[337] Valery A Ugrinovskii. Robust h infinity control in the presence of stochastic uncertainty. *International Journal of Control*, 71(2):219–237, 1998.

[338] Mukhtiar A Unar, David Murray-Smith, and Shahaab Ali Shah. *Design and tuning of fixed structure PID controllers-a survey*. PhD thesis, University of Glasgow, Glasgow, Scotland, UK, 1995.

[339] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[340] Twan Van Laarhoven. L2 regularization versus batch and weight normalization. *arXiv preprint arXiv:1706.05350*, 2017.

[341] Anthony Vannelli and Mathukumalli Vidyasagar. Maximal lyapunov functions and domains of attraction for autonomous nonlinear systems. *Automatica*, 21(1):69–80, 1985.

[342] Richard B Vinter. *Optimal control*. Springer, 2010.

[343] Sofia Visa, Brian Ramsay, Anca L Ralescu, and Esther Van Der Knaap. Confusion matrix-based feature selection. *MAICS*, 710:120–127, 2011.

[344] Ludwig Von Bertalanffy. *General system theory : foundations, development, applications*. George Braziller Inc, 1968.

[345] Athanasios Voulodimos, Nikolaos Doulamis, Anastasios Doulamis, and Eftychios Protopapadakis. Deep learning for computer vision: A brief review. *Computational intelligence and neuroscience*, 2018, 2018.

[346] Dennis Wackerly, William Mendenhall, and Richard L Scheaffer. *Mathematical statistics with applications*. Cengage Learning, 2014.

[347] Stefan Wager, Sida Wang, and Percy Liang. Dropout training as adaptive regularization. *arXiv preprint arXiv:1307.1493*, 2013.

[348] M Waltz and King-sun Fu. A heuristic approach to reinforcement learning control systems. *IEEE Transactions on Automatic Control*, 10(4):390–398, 1965.

[349] Chaofeng Wang, Li Wei, Zhaohui Wang, Min Song, and Nina Mahmoudian. Reinforcement learning-based multi-auv adaptive trajectory planning for under-ice field estimation. *Sensors*, 18(11):3859, 2018.

[350] Fang Wang, Lei Wan, Ye Li, Yu-min Shu, and Yu-ru Xu. A survey on development of motion control for underactuated auv. *Shipbuilding of China*, 51(2):227–241, 2010.

[351] Qing-Guo Wang, Zhiping Zhang, Karl J Astrom, Yu Zhang, and Yong Zhang. Guaranteed dominant pole placement with pid controllers. *IFAC Proceedings Volumes*, 41(2):5842–5845, 2008.

[352] Shikai Wang, Hongzhang Jin, Lingwei Meng, and Guicang Li. Optimize motion energy of auv based on lqr control strategy. In *2016 35th Chinese Control Conference (CCC)*, pages 4615–4620, 2016.

[353] Weiwen Wang and Zhiqiang Gao. A comparison study of advanced state observer design techniques. In *Proceedings of the 2003 American Control Conference, 2003.*, volume 6, pages 4754–4759, 2003.

[354] Xue-Song Wang, Yu-Hu Cheng, and Sun Wei. A proposal of adaptive pid controller based on reinforcement learning. *Journal of China University of Mining and Technology*, 17(1):40–44, 2007.

[355] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[356] Sarah Webb. Deep learning for biology. *Nature*, 554(7693), 2018.

[357] Paul J Werbos. An overview of neural networks for control. *IEEE Control Systems Magazine*, 11(1):40–41, 1991.

[358] Michael E West and Vassilis L Syrmos. Navigation of an autonomous underwater vehicle (auv) using robust slam. In *2006 IEEE Conference on Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control*, pages 1801–1806, 2006.

[359] Cort J Willmott and Kenji Matsuura. Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance. *Climate research*, 30(1):79–82, 2005.

[360] Jonas Witt and Matthew Dunbabin. Go with the flow: Optimal auv path planning in coastal environments. In *Australian Conference on Robotics and Automation*, volume 2008, 2008.

[361] Peter Wlodarczak, Jeffrey Soar, and Mustafa Ally. Multimedia data mining using deep learning. In *2015 Fifth International Conference on Digital Information Processing and Communications (ICDIPC)*, pages 190–196, 2015.

[362] Walter M Wonham. On pole assignment in multi-input controllable linear systems. *IEEE transactions on automatic control*, 12(6):660–665, 1967.

[363] Hui Wu, Shiji Song, Keyou You, and Cheng Wu. Depth control of model-free auvs via reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 49(12):2499–2510, 2018.

[364] QH Wu and AC Pugh. Reinforcement learning control of unknown dynamic systems. In *IEE Proceedings D (Control Theory and Applications)*, volume 140, pages 313–322, 1993.

[365] Yingkai Xia, Kan Xu, Ye Li, Guohua Xu, and Xianbo Xiang. Improved line-of-sight trajectory tracking control of under-actuated auv subjects to ocean currents and input saturation. *Ocean Engineering*, 174:14–30, 2019.

[366] Shengyuan Xu and Tongwen Chen. Robust h-infinity control for uncertain stochastic systems with state delay. *IEEE transactions on automatic control*, 47(12):2089–2094, 2002.

[367] Ru-jian Yan, Shuo Pang, Han-bing Sun, and Yong-jie Pang. Development and missions of unmanned surface vehicle. *Journal of Marine Science and Application*, 9(4):451–457, 2010.

[368] Zheping Yan, Jiyun Li, Yi Wu, and Gengshi Zhang. A real-time path planning algorithm for auv in unknown underwater environment based on combining pso and waypoint guidance. *Sensors*, 19(1):20, 2019.

[369] Xuliang Yao, Xiaowei Wang, Feng Wang, and Le Zhang. Path following based on waypoints and real-time obstacle avoidance control of an autonomous underwater vehicle. *Sensors*, 20(3):795, 2020.

[370] Xue Ying. An overview of overfitting and its solutions. In *Journal of Physics: Conference Series*, volume 1168, page 022022, 2019.

[371] Junku Yuh. Design and control of autonomous underwater robots: A survey. *Autonomous Robots*, 8(1):7–24, 2000.

[372] Silvia M Zanoli and Giuseppe Conte. Remotely operated vehicle depth control. *Control engineering practice*, 11(4):453–459, 2003.

[373] Zheng Zeng, Lian Lian, Karl Sammut, Fangpo He, Youhong Tang, and Andrew Lammas. A survey on path planning for persistent autonomy of autonomous underwater vehicles. *Ocean Engineering*, 110:303–313, 2015.

[374] Enrica Zereik, Marco Bibuli, Nikola Mišković, Pere Ridao, and António Pascoal. Challenges and future trends in marine robotics. *Annual Reviews in Control*, 46:350–368, 2018.

[375] Hongming Zhang and Tianyang Yu. Taxonomy of reinforcement learning algorithms. In *Deep Reinforcement Learning*, pages 125–133. 2020.

[376] Shangtong Zhang and Richard S Sutton. A deeper look at experience replay. *arXiv preprint arXiv:1712.01275*, 2017.

[377] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. Deep learning based recommender system: A survey and new perspectives. *ACM Computing Surveys (CSUR)*, 52(1):1–38, 2019.

[378] Zhen Zhang, Cheng Ma, and Rong Zhu. Self-tuning fully-connected pid neural network system for distributed temperature sensing and control of instrument with multi-modules. *Sensors*, 16(10):1709, 2016.

[379] Kaiyang Zhou, Yu Qiao, and Tao Xiang. Deep reinforcement learning for unsupervised video summarization with diversity-representativeness reward. *arXiv preprint arXiv:1801.00054*, 2017.

[380] Kemin Zhou and John Comstock Doyle. *Essentials of robust control*. Prentice hall Upper Saddle River, NJ, 1998.

[381] Xiaojin Zhu. Semi-supervised learning literature survey. *Comput Sci, University of Wisconsin-Madison*, 2, 07 2008.

[382] Xiaojin Zhu and Andrew B Goldberg. Introduction to semi-supervised learning. *Synthesis lectures on artificial intelligence and machine learning*, 3(1):1–130, 2009.

[383] Pavel Zítek, Jaromír Fišer, and Tomáš Vyhlídal. Dimensional analysis approach to dominant three-pole placement in delayed pid control loops. *Journal of Process Control*, 23(8):1063–1074, 2013.

DOCTORAT BRETAGNE LOIRE / MATHSTIC

Titre : Contributions au développement de contrôleurs d'AUV basés sur de l'apprentissage profond par renforcement

Mot clés : Véhicule Sous-marin Autonome, Contrôleur, Apprentissage Profond par Renforcement, Suivi de Points de Repère, Soft Actor-Critic, Proportionnel-Intégral-Dérivé

Résumé : L'environnement marin est un cadre très hostile pour la robotique. Il est fortement non-structuré, très incertain et inclut beaucoup de perturbations externes qui ne peuvent pas être facilement prédites ou modélisées. Dans ce travail, nous allons essayer de contrôler un Véhicule Sous-marin Autonome (AUV) afin d'effectuer une tâche de suivi de points de cheminement, en utilisant un contrôleur basé sur de l'apprentissage automatique. L'apprentissage automatique a permis de faire des progrès impressionnants dans de nombreux domaines différents ces dernières années, et le sous-domaine de l'apprentissage profond par renforcement a réussi à concevoir plusieurs algorithmes très adaptés au contrôle continu de systèmes dynamiques. Nous avons choisi d'implémenter l'algorithme du Soft Actor-Critic (SAC), un algorithme d'apprentissage profond par renforcement régularisé en entropie permettant de simultanément remplir une tâche d'apprentissage et d'encourager l'exploration de l'environnement. Nous avons comparé un contrôleur basé sur le SAC avec un contrôleur Proportionnel-Intégral-Dérivé (PID) sur une tâche de suivi de points de cheminement et en utilisant des métriques de performance spécifiques. Tous ces tests ont été effectués en simulation grâce à l'utilisation de l'UUV Simulator. Nous avons décidé d'appliquer ces deux contrôleurs au RexROV 2, un Véhicule Sous-marin Téléguidé (ROV) de forme cubique et à six degrés de liberté converti en AUV. Grâce à ces tests, nous avons réussi à proposer plusieurs contributions intéressantes telles que permettre au SAC d'accomplir un contrôle de l'AUV de bout en bout, surpasser le contrôleur PID en terme d'économie d'énergie, et réduire la quantité d'informations dont l'algorithme du SAC a besoin. De plus nous proposons une méthodologie pour l'entraînement d'algorithmes d'apprentissage profond par renforcement sur des tâches de contrôle, ainsi qu'une discussion sur l'absence d'algorithmes de guidage pour notre contrôleur d'AUV de bout en bout.

Title: Contributions to the development of Deep Reinforcement Learning-based controllers for AUV

Keywords: Autonomous Underwater Vehicle, Controller, Deep Reinforcement Learning, Waypoint Tracking, Soft Actor-Critic, Proportional–Integral–Derivative

Abstract: The marine environment is a very hostile setting for robotics. It is strongly unstructured, very uncertain and includes a lot of external disturbances which cannot be easily predicted or modelled. In this work, we will try to control an Autonomous Underwater Vehicle (AUV) in order to perform a waypoint tracking task, using a machine learning-based controller. Machine learning allowed to make impressive progress in a lot of different domain in the recent years, and the subfield of deep reinforcement learning managed to design several algorithms very suitable for the continuous control of dynamical systems. We chose to implement the Soft Actor-Critic (SAC) algorithm, an entropy-regularized deep reinforcement learning algorithm allowing to fulfill a learning task and to encourage the exploration of the environment simultaneously. We compared a SAC-based controller with a Proportional-Integral-Derivative (PID) controller on a waypoint tracking task and using specific performance metrics. All the tests were performed in simulation thanks to the use of the UUV Simulator. We decided to apply these two controllers to the RexROV 2, a six degrees of freedom cube-shaped Remotely Operated underwater Vehicle (ROV) converted in an AUV. Thanks to these tests, we managed to propose several interesting contributions such as making the SAC achieve an end-to-end control of the AUV, outperforming the PID controller in terms of energy saving, and reducing the amount of information needed by the SAC algorithm. Moreover we propose a methodology for the training of deep reinforcement learning algorithms on control tasks, as well as a discussion about the absence of guidance algorithms for our end-to-end AUV controller.