POLITECNICO DI TORINO

Master of Science Degree in
MECHATRONIC ENGINEERING

MASTER THESIS

Visual based local motion planner with Deep Reinforcement Learning

---

Supervisor

Prof. Marcello CHIABERGE

Candidate

Mauro MARTINI

s260771

OCTOBER 2020

Abstract

This thesis aims to develop an autonomous navigation system for indoor scenarios based on Deep Reinforcement Learning (DRL) technique.

Autonomous navigation is a hot challenging task in the research area of robotics and control systems, which has been tackled with numerous contributions and different approaches. Among them, learning methods have been investigated in recent years due to the successful spreading of Artificial Intelligence and Machine Learning techniques. In particular, in reinforcement learning an agent learns by experience, i.e. through the interaction with the environment where it is placed, avoiding the need of a huge dataset for the training process.

Service robotics is the main focus of the research at PIC4SeR (PoliTo Interdepartmental Centre for Service Robotics), where the idea of this thesis project is born as part of a broader project. Under the supervision of Professor M. Chiaberge (PoliTo), member of the group, the thesis embraces the vision of the centre, which is to develop high-tech solutions for peculiar fields such as precision agriculture, surveillance and security, in addition to assist people in their every-day life. As a matter of fact, an autonomous navigation system enables competitive advantages in a wide variety of the applications of interest.

Deep Deterministic Policy Gradient (DDPG) is the specific DRL algorithm applied to train an agent in a simulated environment using ROS (Robot Operating System). Training simulations offer different types of scenarios presenting both static and moving obstacles. The main goal of the project is to provide a safe collision-free navigation in an unknown indoor environment. An Artificial Neural Network (ANN) is used to directly select suitable actions for the robot, expressed in terms of linear and angular velocity (ANN output). Input information is composed of robot pose and goal position, in addition to raw images provided by a depth camera. A great focus is also devoted to reduce the computational cost of the model in the training phase, as well as the energy consumption in a potential hardware implementation. For this reason, an efficient architecture of the CNN is studied, paying attention to both desired performances and costs. Firstly, a set of convolutional layers is needed to extract high-level features from depth images. Then fully-connected layers predict the action for the robot. Beside these aspects, also sensor data play a key role in a navigation system. From a research point of view, it is interesting to evaluate the performance of the algorithm when using depth images, compared to other popular implementations based on LiDAR sensor. On the one hand, a camera offers a rich depth information. On the other hand a simple 2D LiDAR is able to cover a wider field of view.

The navigation system has been tested in a virtual environment with obstacles. Despite the difficulty of the challenge and the amount of resources required for the development, the system can be considered a good starting point for future works. The implementation of the algorithm on a real robot will be a natural next step for the project.

Contents

List of Tables

List of Figures

Principal Acronyms

AI

Artificial Intelligence

ML

Machine Learning

ANN

Artificial Neural Network

CNN

Convolutional Neural Network

RL

Reinforcement Learning

DRL

Deep Reinforcement Learning

DDPG

Deep Deterministic Policy Gradient

ROS

Robot Operating System

LiDAR

Light Detection and Ranging

Chapter 1

Introduction

1.1 Objective of the thesis

Autonomous navigation is a challenging task in the research area of robotics, which has been tackled with numerous contributions and different approaches. Among them, learning methods have been investigated in recent years due to the successful spreading of Deep Learning, a recent approach to Artificial Intelligence. In particular, in Reinforcement Learning an agent learns by experience, i.e. through the interaction with the environment where it is placed, without the need of a huge dataset.

The idea of the thesis is born at the PIC4SeR (PoliTo Interdepartmental Centre for Service Robotics), as part of a broader project focused on service robotics. The work embraces the vision of the centre, which is to develop high-tech solutions for peculiar fields of application such as precision agriculture, smart city, surveillance and security, in addition to assist people in their every-day life as well as in emergency situations. As a matter of fact, an autonomous navigation system enables competitive advantages in a wide variety of the applications of interest. In particular, the development of the system will be mainly devoted to indoor service and assistance purposes for people.

Deep Deterministic Policy Gradient (DDPG) is the specific Deep Reinforcement Learning (DRL) algorithm applied to train an agent in a virtual environment using ROS (Robot Operating System). The main goal of the project is to provide a safe collision-free navigation in an unknown indoor scenario. A Convolutional Neural Network (CNN) is used to directly select suitable actions for the robot, expressed in terms of linear and angular velocity (CNN output). Input information is composed of robot pose and goal position, in addition to images provided by an Intel RealSense

depth camera. Depth images are single channel grey-scale images which provide distance information. A great focus is also devoted to minimize the computational cost of the model, looking forward to a future hardware implementation. From a research point of view, it is interesting to evaluate the performance of the navigation system when using depth images, compared to a different popular implementation based on LiDAR sensor. On the one hand, a camera offers a rich depth information. On the other hand a simple 2D LiDAR is able to cover a wider field of view.

1.2 Organization of the thesis

An overview of the composition of the thesis is given, briefly describing the content of each chapter.

Chapter 1 introduces the reader to the goal of the thesis. The main aspects of the methodology carried out in the work are also summarized.

Chapter 2 starts framing the problem of navigation. Then, an overview of the state of the art of autonomous navigation systems for robotics with Deep Reinforcement Learning is proposed. The role of the Depth Camera in robotics works is also depicted.

Chapter 3 illustrates the historical foundation and the main concepts of Machine Learning and Deep Learning. It is convenient to provide the reader with a small theoretical background about Artificial Neural Networks for a easier understanding of the framework and results of the project.

Chapter 4 is devoted to Reinforcement Learning. The key elements of this learning framework are first set. Then, a collection of the most popular algorithms and methods is listed. In the final section of the chapter, the Deep Deterministic Policy Gradient algorithm is described more in detail, as it is the one used in the project.

In chapter 5 the definition of robotic platform is given, analysing the main components of the TurtleBot3 platform. Moreover, the software tools and the sensors used for the simulation are depicted.

Chapter 6 comprises a thorough description of the implementation of the algorithm in ROS. The Artificial Neural Networks designed for both a LiDAR-based and a camera-based navigation are illustrated, together with the whole training process.

Chapter 7 is devoted to the testing phase in a virtual scenario with obstacles. Results are reported and discussed. A comparison between the LiDAR-based navigation and the visual one is carried out. Conclusions and suggestions for future works are finally given.

Chapter 2

State of the art

2.1 Introduction to navigation

Navigation in robotics refers to the ability of a mobile robot to move from its starting position to a desired destination, by selecting a valid path composed of a sequence of configurations.
The possibility to navigate autonomously in an environment is an essential and challenging problem in robotics. A wide variety of approaches has been developed, making also possible to classify the existing systems according to their main features. First of all, it is useful to briefly describe the three main elements needed by a navigation system in an indoor scenario, which are:

1. a *localization system* to identify the robot position and orientation with respect to a reference frame;
2. a *path planner* to compute a suitable sequence of configurations for the robot to reach the goal;
3. a *motion controller* to select the actions for the robot to make it follow the desired computed trajectory.

Given the target position, the robot should be able to localize itself and plan a suitable path in the indoor environment. Then, the motion controller will be responsible of moving the robot trying to fit precisely the points on the trajectory. It is possible to distinguish between map-based and mapless navigation tasks. The majority of the traditional methods need to map the environment where the robot moves before being able to accomplish the localization and the path planning tasks. Simultaneous Localization and Mapping (SLAM) is one of the principal technique for localization, which simultaneously allow to build the map of the surrounding

environment. However, it is often tough in multiple real scenarios to build a precise map. Furthermore, complex environments with obstacles of many shapes represent a huge limitation for traditional methods. Computational costs and versatility are key aspects for an autonomous navigation system. Versatility can be defined as the ability to generalize the performance with respect to different situations regardless of the commands designed by the programmers.
To this extent, Deep Reinforcement Learning is an interesting mapless solution to increase both flexibility and autonomy. Moreover, the system architecture results to be simpler with DRL, since the path planner and the motion controller collapse in a single decision-making entity.

2.2 DRL for autonomous navigation

Deep Reinforcement Learning emerged as a potential approach for robotic navigation only in recent years. It presents multiple advantages with respect to both traditional methods and others machine learning technique for what specifically concerns the navigation task. An introduction to deep reinforcement learning concepts will be presented in the following chapters. Here, a selection of the principal publications from 2017 to 2020 representing the state of the art of DRL applications for robotic navigation is shortly reported. [1], [2], [3], [4] and [5] exploit DRL by changing the learning model and algorithm and by using different sensor data. A brief description of the content of each work is proposed, highlighting the peculiarity of each approach.

"Virtual-to-real Deep Reinforcement Learning: Continuous Control of Mobile Robots for Map-less Navigation"

This work has been published in 2017. It proposes a basic but robust implementation of a deep-RL agent to guide a wheeled robot in an unseen real environment. The navigation has been tackled with a mapless motion planner, trained from scratch with a continuous control deep-RL algorithm. In particular, an asynchronous DDPG, called ADDGP, has been used. The algorithm exploits only 10 sparse range laser points. Although it is intended as a low-cost solution, the system shows great responses in new environments with respect to map-based methods thanks to reinforcement learning.

"An End-to-End Deep Reinforcement Learning-Based Intelligent Agent Capable of Autonomous Exploration in Unknown Environments"

This work has been published in 2018. The project aims to realize an end-to-end obstacle avoidance and navigation system based on DRL. The authors chose to

develop a Memory-Based Deep Reinforcement Learning algorithm. They show the importance of using a continuous action space to improve robot's performance and the advantages of training and testing in simulation. Moreover, a sensor fusion technique is exploited to improve the noisy depth information thanks to additional range sensors.

"Collision Avoidance for Indoor Service Robots through Multimodal Deep Reinforcement Learning"

This paper has been published in 2019. The work presents an innovative artificial neural network model trained with a DDPG algorithm to get a reliable collision avoidance algorithm for indoor service robotics. In particular, the authors introduced a set of long short-term memory (LSTM) layers. The model is able to independently extract features from different sensors and combine them to select an action for the robot. Moreover, an interesting analysis about how to cover the gap between simulation and real world is conducted through depth images processing during training.

"Learning Navigation Behaviors End-to-End with AutoRL"

This paper has been published in 2019. The authors approaches the development of an end-to-end learning algorithm for navigation with an innovative technique: AutoRL. It is a new automation trend in RL that automatically takes care of the neural network architecture and of the reward by applying hyper-parameter optimization. The system has been tested both in simulation and in reality and shows promising results with respect to classical methods.

"Goal-Oriented Obstacle Avoidance with Deep Reinforcement Learning in Continuous Action Space"

This paper has been published in February 2020 and it is the most recent contribution to this thesis in literature. The work aims to reach a high quality performance in obstacle avoidance, also when dealing with objects of complex shapes. For this purpose the proposed DDPG network controls the action of the robot exploiting a depth-wise separable convolution, receiving in input a stack of successive depth images. By using images of the surrounding scene at previous time instants it is possible to compensate the limited field of view of depth cameras.

2.3 Depth Camera for Robotic Applications

Depth Camera and depth images are increasing their relevance in robotics in almost every field of application. Depth cameras became a popular sensor when in 2010 Microsoft released its first Kinect for the Xbox gaming platform. This technology combines RGB cameras as well as infrared sensors, which allow for real-time gestures recognition. From the robotic point of view, the interest for Kinect sensors is related to the great trade-off of speed and resolution offered for 3D perception. They resulted to be also a cheaper solution compared to existing technology such as laser range finders and stereo vision systems. The advantages of depth cameras are explored in several works about robotic navigation published around 2012. [6] is a work published in 2011 which show an application of depth camera (Kinect) for indoor robot localization and navigation. [7] also proposed in 2012 a real-time navigation system for a humanoid robot based on depth camera data. More recent depth camera models have increased its capability and diffusion. For example, the Intel RealSense is used in a great amount of works. Among them [5] in 2020 uses a D435 Intel RealSense to enable a robot navigation completely based on depth images and DRL.

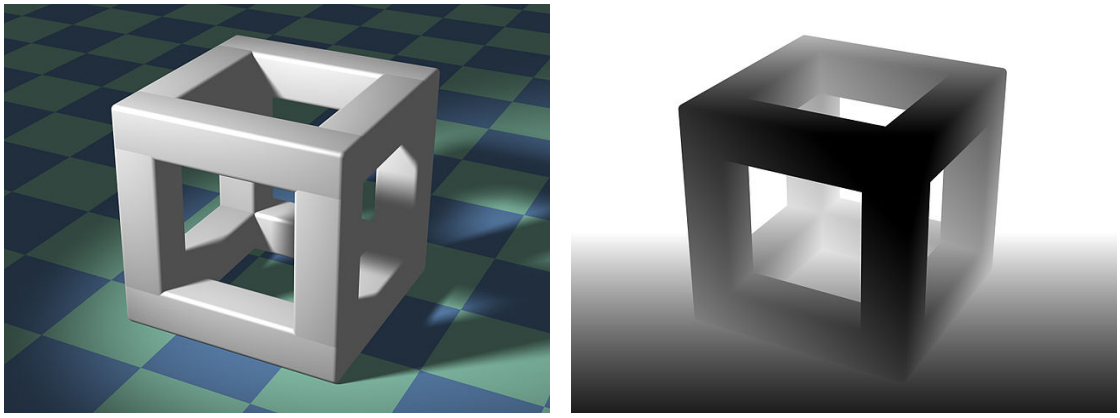

Figure 2.1: Example of depth image in a virtual environment: on the left the original image, on the right the depth map based on the distance from the camera. [8]

In 2012, also deep learning spread out thanks to the great results of convolutional neural network on image classification challenges, above all on the ImageNet dataset. In the field of robot perception and computer vision convolutional neural network are a standard tool to process RGB images and extract features from them. However, for robotic tasks consisting in a physical interaction with objects, such

as grasping or obstacle avoidance, RGB images may be not necessary. In fact, for those tasks the key information are mainly related to geometry, pose and other color independent features. Depth images are single channel grey-scale images which are able to provide distance information with a reduced computational cost. For this reason, in recent years depth images are spreading out in robotic applications. The article at [9] suggests three interesting examples. Among them, [10] proposed a robot bed-making which is able to combine effectively depth images, processed with very simple operations, and RGB images for corners detection. An additional advantage of depth images is that it is possible to fix a 'blackout' threshold, to remove high distance section of the image and let the algorithm focus on relevant information. This perfectly shows the effectiveness of depth images for complex robotic tasks.

An ongoing research trend consists in estimating depth from monocular RGB images through deep learning. Such a possibility is especially explored by Google research groups, which published two works in 2019. The first one focused on estimating depth from wild RGB videos, the second one on moving people. Further information can be found at [11] and [12]. On the one hand depth estimation would provide a great reduction of sensor cost, making a simple RGB camera the only hardware needed for perception. On the other hand the efficiency of depth estimation have to be sufficiently high to fit with a real-time navigation algorithm.

Chapter 3

Machine Learning

3.1 Chapter overview

This chapter aims to introduce the reader to Machine Learning (ML) concepts. This is a first necessary step to understand the Deep Reinforcement Learning technique used in the thesis. The first part of the chapter contains an high level discussion about machine learning and deep learning, trying to make the point about their definitions and also to provide some historical notes. The second section is devoted to a brief explanation of the basic mathematical models and concepts behind ML, with the aim of giving the reader an idea of what is an artificial neural network and how it works.

3.2 AI, Machine Learning and Deep Learning

In the digitalization era, it is very common to hear talking about Artificial Intelligence. However, often the meaning of the terms Artificial Intelligence (AI) and Machine Learning (ML) is confused by people who are not experts of the field. Without any doubt, the concept of AI and ML, as well as of Deep Learning (DL), are related by the evolution of the technological sector in time. AI appears for the first time in the 1950s. The term ML was first used by A.Samuel in 1959 [13], whilst DL was born recently. The mutual relation between the three of them, can be described with concentric circles, as shown in Figure 3.1. Hence, AI is a broader concept. It is possible to refer to AI as the set of operations performed by computers which are able to accomplish tasks usually associated with intelligent beings. Differently, ML can be identified as a sub-field or methodology of AI, with the aim of learning automatically from data. Then, DL is a particular branch

derived from ML, which is characterized by the usage of 'deep' artificial neural networks, i.e. learning models presenting a great number of layers and neurons. [14]

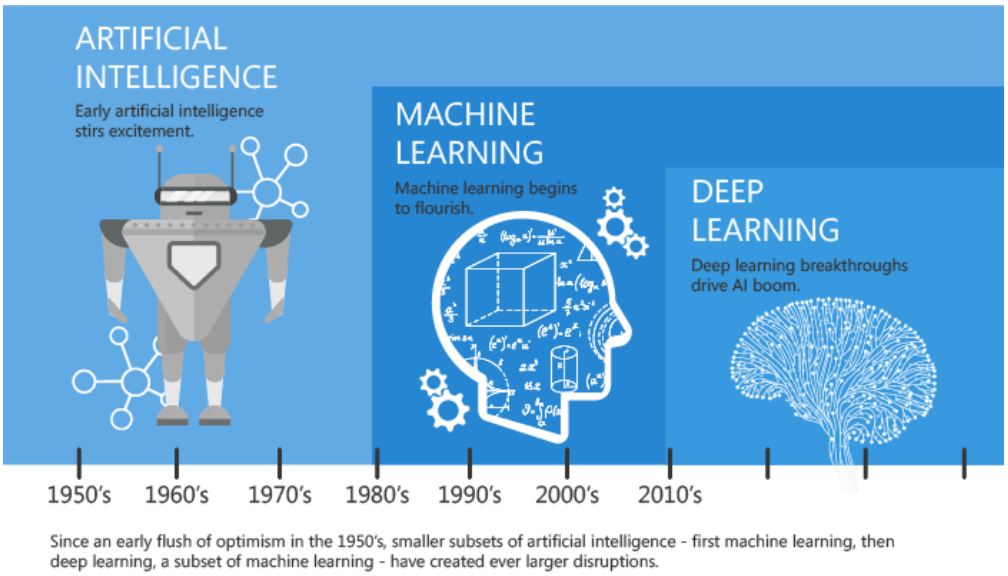

Figure 3.1: Artificial Intelligence, Machine Learning and Deep Learning. [15]

3.3 History of Deep Learning

The first wave of research about machine learning produced very simple models of a neuron. This series of attempts of reproducing the brain function is also known as *cybernetics*. The very first model is attributed to McCulloch and Pitts in 1943, who modeled a neuron with a simple linear binary classifier. By checking the sign of the function $f(x, w) = x_1 w_1 + ... + x_n w_n$ it was able to distinguish between two classes of inputs. The weights needed to be set correctly by the human. An evolutionary step comes in the 1950s with the Rosenblatt's perceptron which was able to adjust the weights thanks to a first idea of an iterative 'training process' that exploit a set of example inputs for each class. The perceptron algorithm had a great success. In these years Alan Turing created a test to verify if a machine could bring a person to believe to be talking with another human. Another important contribution to machine learning comes from the Adaptive Linear Element (ADALINE) by Widrow and Hoff in 1960. The algorithm used to change the weights of the ADALINE was a fist version of the *stochastic gradient descent*, which is one of the principal training algorithms also in modern deep learning. Linear models are still widely used and reinterpreted today, although they present limitations. For example, it is

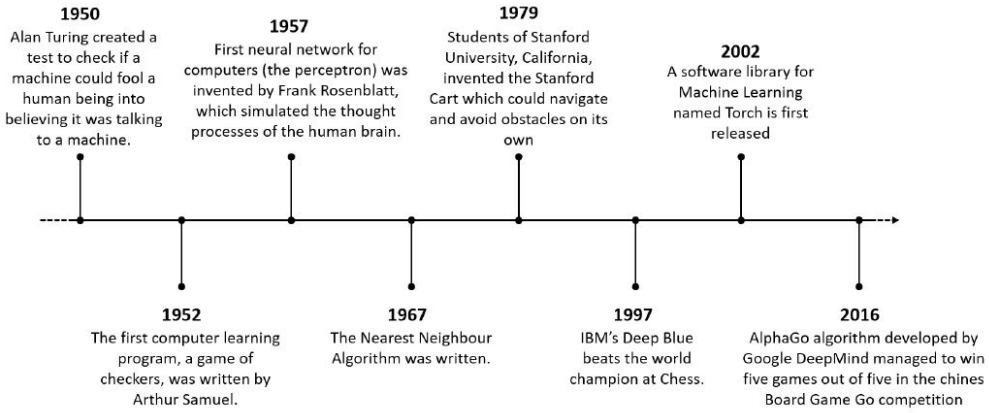

Figure 3.2: Time-line of deep learning history.

known that they have problems in learning the XOR function. In 1969 Minsky and Papert exposed these negative considerations in their paper. This brought to a loss of interest in learning linear models inspired by neurons in the following period, called the 'first winter' of AI.
The second period of active research about neural networks arose in the 1980s guided by the *connectionism* movement. Fundamental contributions to deep learning were conceived during the connectionism. The first one consists in the idea of *distributed representation*. According to this, inputs should be represented by many shared features. The framework for artificial neural networks was set with the idea of a model for Parallel Distributed Processing (PDP), exploiting multiple connected units (neurons). Rumelhart and McClelland were the principal contributors to these ideas inspired by the human brain. Together with Williams they also conceived the concept of hidden layers and the back-propagation algorithm, which is the actual dominant method to train deep learning models. The second research wave proceeded till the mid-1990s. Long short-term memory (LSTM) networks were another important results obtained in those years by Hochreiter and Schmidhuber to model long sequences of information. Different groups of people in the world kept alive the research on neural networks obtaining also great results in some cases. Among them Yoshua Bengio,Yann LeCun, Geoffrey Hinton. However, the evident difficulty in training deep models stopped the excitement of the majority of researchers.

From 2006, the combination of improvements in the model efficiency and new computational possibilities brought deep neural networks outperform other ML models. The term 'deep' learning was chosen to emphasize the successful architecture of neural networks. This third research wave is still ongoing, focusing on the ability of generalize well also from small dataset or with unsupervised learning techniques,

i.e. without the usage of labels for data.
The recent breakthrough of deep learning can be motivated considered some key factors. First of all, in the "Big Data" era, huge datasets are available to train deep models, which were computationally prohibitive in the past. A rule of thumb suggests that a dataset of 10 millions of element is enough to reach the human ability to classify items of different classes. Very popular are the MNIST dataset of handwritten numbers or the ImageNet dataset. This last one contains more than 14 millions of images of about 20.000 different categories. The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) organized around the dataset creation from 2010, had a strong impact on deep learning evolution. In 2012, a convolutional neural network called AlexNet obtained a top-5 error of 15.3%, which was an incredible result compared to previous attempts [16]. In the following years other models such as VGG, GoogleNet and ResNet have overcome also this performance.
Another fundamental factor for deep learning success has been the spreading of powerful computational hardware such as fast CPUs and general purpose GPUs. Today, GPU are build with new architectures providing optimized parallel operations, often dedicated to machine learning purposes. This is a trivial but key aspect, since a high number of neurons is crucial for many applications. Indeed, looking at the intelligent organism in nature, the number of biological neurons increases with the level of intelligence: humans have about 10^11 neurons, rabbits have around 10^8 while jellyfish have slightly more than few thousands neurons. Today, big artificial neural networks present about 10^6 neurons. Moreover, the number of connections per neuron was initially limited by the hardware. Then, it became a design choice. Biological neural networks present sparse connections, which is important for artificial models to be able to reach their performance with limited resources. Human neurons have about 10^4 connections, which is not an exorbitant quantity if we look at some neural network models.

3.4 Machine Learning concepts

In this section a brief description of Machine Learning concepts is proposed. The goal is to provide the reader with a basic understanding of how neural networks works and which are the main strategies and difficulties related to their implementation.

3.4.1 The artificial neuron: Threshold Logic Unit (TLU)

From the historical evolution of deep learning discussed in the previous section, it is clear that the concept of artificial neural networks has been inspired by biological models of the human brain. Although it has to be pointed out that the evolution of modern machine learning mainly relies on mathematics, statistics and numerical optimization, neuroscience should also be considered as a crucial source of inspiration. A schematic model of the biological neuron in reported in Figure 3.3.

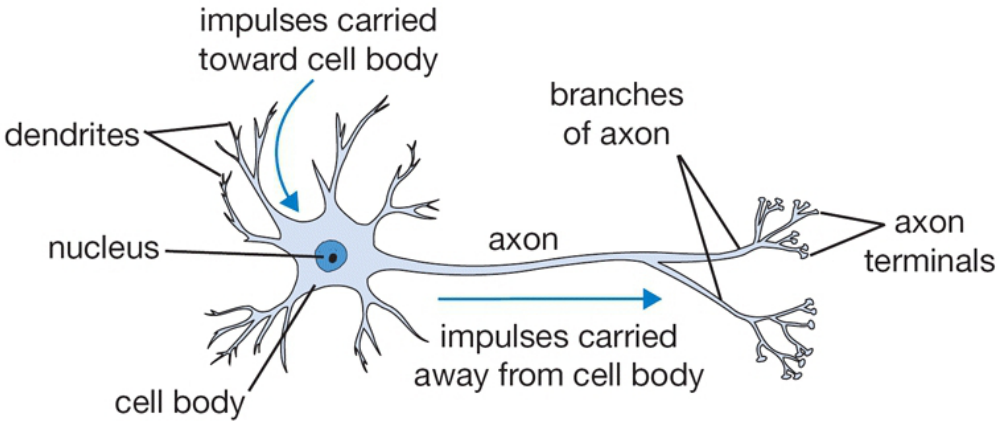

Figure 3.3: Biological neuron: a schematic model.
[17]

The main element which compose a biological element are: [18]

- *Cell body or Soma*: it encloses the nucleus of the nerve cell;
- *Dendrites*: they are branched extensions which allow the cell body to receive input signals from neighboring neurons;
- *Axon*: it is a particular and unique extension connecting the cell body to the synapses. It is responsible of carrying the electrical signal;
- *Synapses*: they are a ramified structure located at the final section of the axon. They pass information to the other neurons.

The connection of multiple neurons through dendrites and synapses, as shown in Figure 3.3, results in a biological neural network. Roughly speaking, neural activity is based on the flow of electrical signals from a neuron to the neighboring ones. More in detail, the signal flow is governed by electrochemical processes such as voltage-gated ion exchange to let the electrical signal move through all the neuron's cell. To make this process clear, let's try to follow the entire path of a signal. If the signal is passing through the axon termination, it will be transmitted to synapses. Here, a certain amount of *neurotransmitters* is released. This chemical substance has a fundamental influence on the synapse *conductivity*. Its quantity can attenuate or boost the signal. In other words, it acts like a weight. Once passed the synaptic junction, the signal is forwarded to the post-synaptic neuron thanks to dendrites, which are able to capture neurotransmitters. Local small currents are created by the positive and negative signals arriving from the dendrites in the soma. Here, they can be mixed together and sum up. Finally, when the electrical potential in the soma reaches a certain threshold, an impulse is generated and transmitted again along the axon.

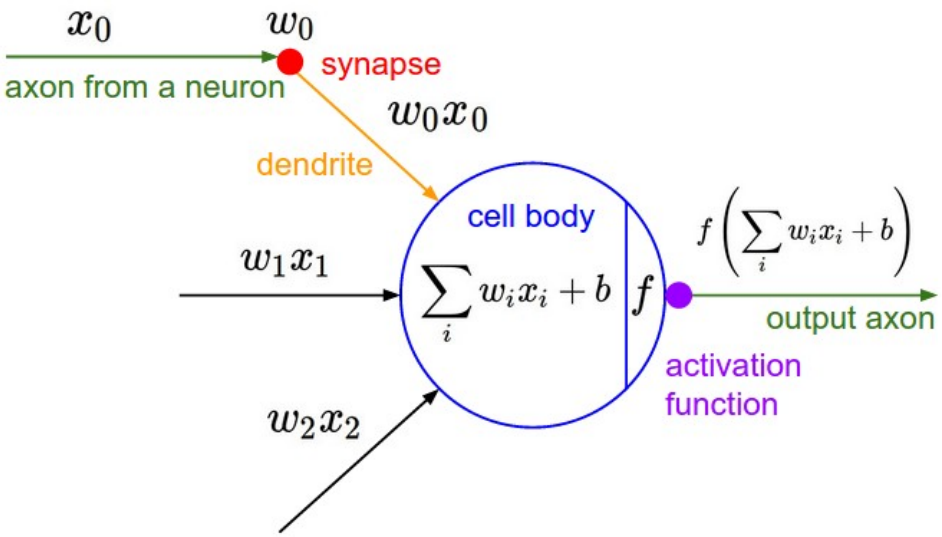

Figure 3.4: The artificial neuron model with a generic activation function. [17]

By looking at a simple artificial neuron model (see Figure 3.4), the Threshold Logic Unit (TLU), a parallelism with the described process is straightforward. Starting back from the axon, a numerical signal between 0 and 1 becomes the input signal for another neuron through the synapses. Here, a weight w_i is assigned according to the conductivity level of the synapses. All the weighted input signal (from the dendrites) get sum as it happens in the soma. Finally, the signal is forwarded

depending on a specific activation function. A basic step activation function is namely a threshold. It is important to highlight that not all the connections between synapses are equally weighted. They depend on their priority, and can be excited or inhibited according to the amount of chemical transmitter. The same happens in general for artificial neural networks (ANNs) with weights, but also thanks to inhibitory signals.
Moreover, according to the McCullogh-Pitts first model of an artificial neuron, the following assumptions are true:

- the activation function of each neuron is an established threshold *theta*;
- the output is binary (logic unit);
- input signals are identically weighted and can be inhibited ;
- at each time step the output signal will be equal to 1 if the sum of all the weighted inputs is greater than the threshold and the neuron is not inhibited, 0 otherwise.

Thus, the behaviour of the artificial neuron can be represented with the following function.

$$output = \begin{cases} 1 : \sum_{i=1}^{n} w_i x_i \geq \theta \wedge no\ inhibition \\ 0 : otherwise \end{cases}$$

3.4.2 Perceptron

Perceptron is a supervised algorithm which allows to learn a binary classifier. Rosenblatt's perceptron is the historical evolution of TLU model for artificial neuron, with the only following differences:

- neurons have positive or negative weights and bias, all different;
- there is no inhibitory signals;
- the activation function is still a binary step in the classic implementation, but the output can assume values [−1,1] instead of [0,1];
- it has a learning algorithm.

By expressing the threshold with a bias term, the model can be reshaped as follow:

$$f(\mathbf{x}) = \begin{cases} 1 : \sum_{i=1}^{n} w_i x_i + b > 0 \\ -1 : otherwise \end{cases}$$

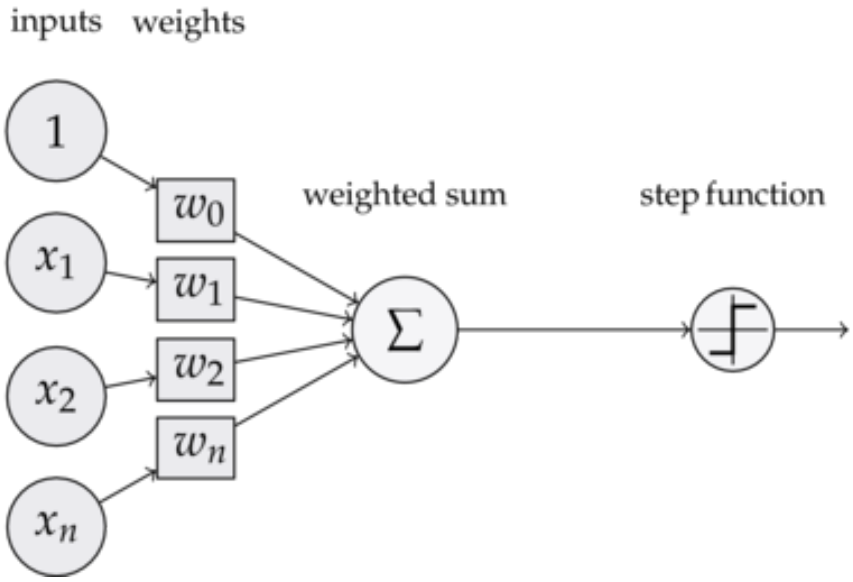

Figure 3.5: Perceptron schematic.

They key concept is that the goal of this model is to classify samples (vectors) belonging to two different categories. With the described formulation, the classifier is able to learn how to divide only a linearly separable set of points. Linear classifier can be also combined to be applied on multi-categories problems. Nonetheless, the most important innovation of the perception is its online learning algorithm, which is used to adjust the weights of the model for each training sample. The iterative process is roughly described in Algorithm 1, where y_i are the sign labels of training samples x_i.

Algorithm 1 Perceptron algorithm

Initialize w = 0 , b = 0 ▷ weights and bias are set to 0
repeat
  if $y_i[w_i x_i + b] \leq 0$ **then**
    $w \leftarrow w + y_i x_i$
    $b \leftarrow b + y_i$
  end if
until all classified correctly

Hence, f(**x**) is used to classify the item **x** as positive or negative. At the end of the process the correct weights for the linear binary classifier are obtained. It can be represented graphically as a *decision boundary*. The position of the decision boundary with respect to the origin is shifted by the bias term. An example is reported in Figure 3.6 for a 2 dimensional case.

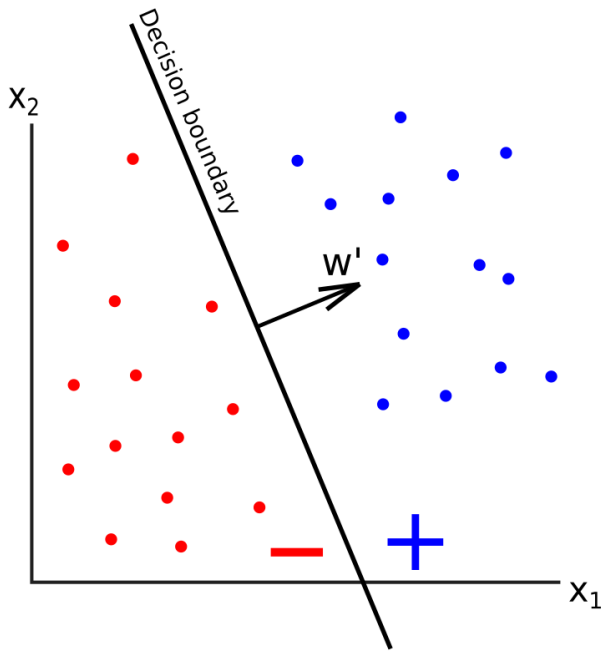

Figure 3.6: Example of decision boundary for a binary classification problem.

3.4.3 Architecture of Artificial Neural Networks

As explain for the perception, linear classifiers are not able to deal with non-linear problems. One of the main historical example is that they cannot learn the XOR function. For this purposes, the evolutionary architecture of artificial neural network (ANNs) resulted to be successful. As shown in Figure 3.7, a generic structure is obtained by connecting single neurons with a precise organization. Neurons vertically aggregated compose what is called a *layer*. By looking at the schematic, the first layer on the left is also called *input layer*, since its neurons are directly connected to input signals. On the opposite side, there is the *output layer*. In the specific case of Figure 3.7 it is composed by a single neuron. It is responsible of labeling the numerical signals arriving from previous layers, i.e. of the final classification of the output. In the middle, there are the *hidden layers*.

To sum up, the main element of a basic neural network architecture as the one considered are:

- the input signals x_i;
- the weights w_{ij} and bias b_j of each connection;
- the activation function a_j of each neuron;

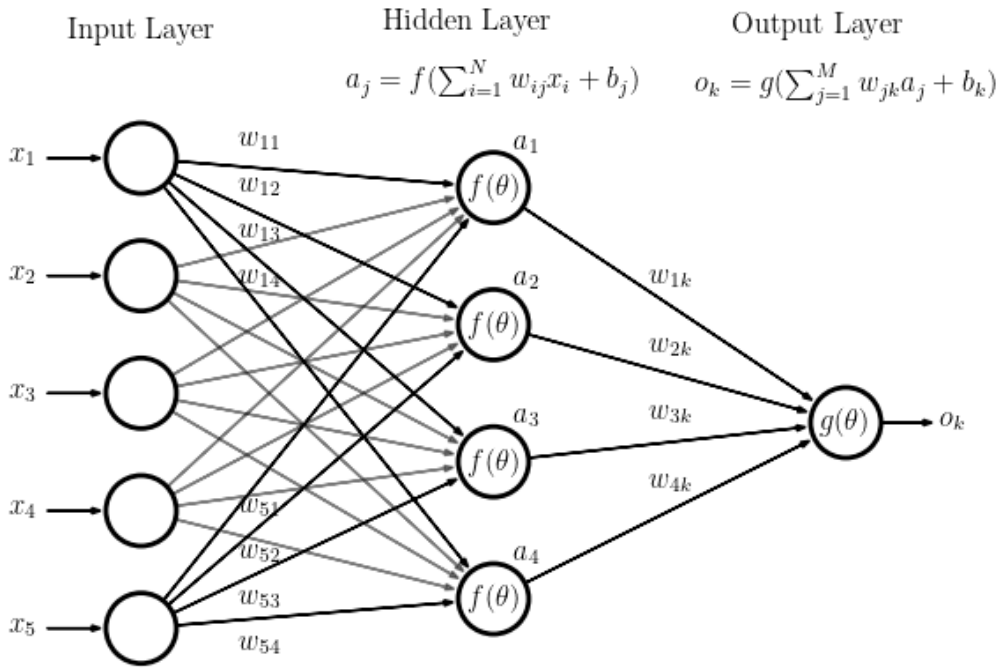

Figure 3.7: Example of neural network architecture.
[19]

The weights decrease or boost the numerical signal of each connection. Once it arrives at the neuron, the activation function decide to *fire* the signal or not. It is possible to notice that the activation functions used in the hidden and in the output layers are usually different, according to the specific function to learn. Moreover, with respect to the perceptron, they can have different expressions that can be generally written as:

$$a_j = \sigma(\sum_{i=1}^{N} w_{ij}x_i + b_j)$$

3.4.4 Activation functions

The learning process of a neural network consists in the adaptation of its weights and bias. However, when using an activation function with binary output, as done in the perceptron, small changes of the parameters may cause a significant modification of the output. In other words, the output can switch from 0 (or -1 according to the activation function used) to 1, with a small variation of the weights.

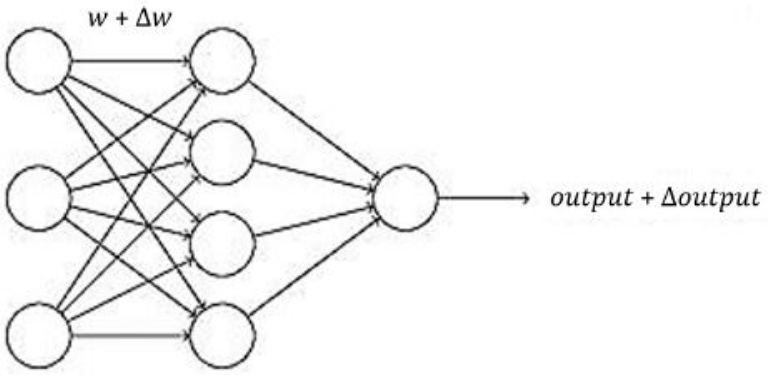

Figure 3.8: Neural network: propagation of the weight's variation till the output.

Sigmoid function

The solution to overcome this limitation is related to the introduction of the *sigmoid function*.

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

A more specific formulation for neurons will be:

$$\sigma(wx + b) = \frac{1}{1 + exp(-\sum_{i=1}^{n} w_i x_i - b)}$$

It provides a smoother variation of the output with respect to modification of the parameters. This mitigates the learning process with respect to step functions. Nonetheless, the output still remain bounded in a limited range. Today the sigmoid activation function is widely used in machine learning.

Linear activation function

A linear activation function of the form $\sigma(z) = cz$, produces an output proportional to the input. Graphically, it results in a simple line passing for the origin. Although the output is not binary, it produces some problems in neural network's training. In fact, in a network having all neurons with linear units the signal resulting from all the connections will merely be a linear signal as well. It is easy to understand it if we remember that the output of a neuron will be a weighted input for the following one. The badness of this consideration relies on the fact that it is possible to replace multiple layers with just an equivalent one.

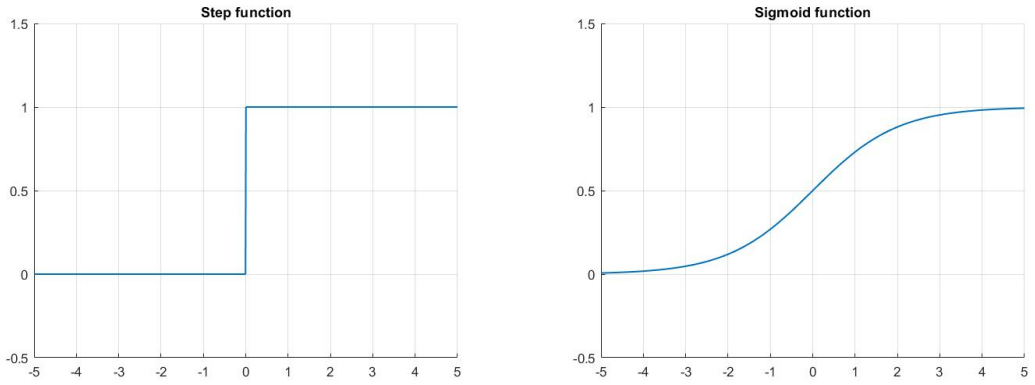

Figure 3.9: On the left a step activation function: the output is binary as in the perceptron due to the sharp variation from 0 to 1. On the right the sigmoid activation function showing its smooth trend in the same interval.

Tanh activation function

The hyperbolic tangent activation function presents a behaviour similar to sigmoid functions. The main difference between the two of them is the range of output values. Indeed tanh(z) presents an output bounded within -1 and 1. The hyperbolic tangent can be preferred to the sigmoid for reasons strictly related to the specific application. The expression of the function is reported below, whilst its graphical representation is shown in Figure 3.10.

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

The expression for the activation function of the neurons considering $tanh(wx + b)$ can be formulated with:

$$\tanh(z) = \frac{1 + tanh(z/2)}{2}$$

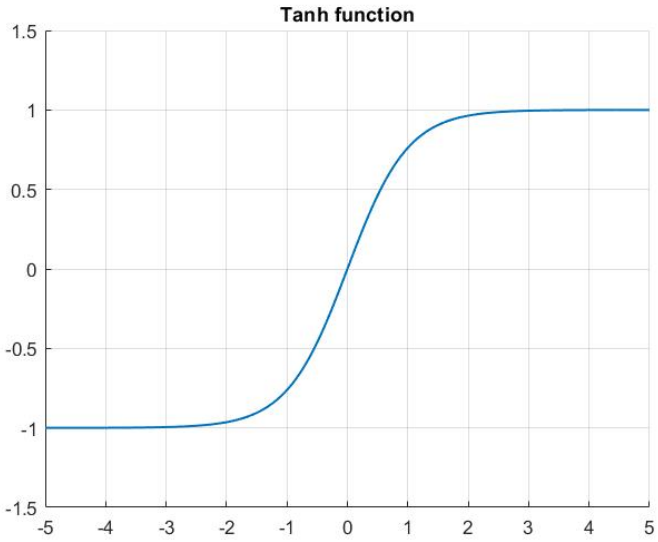

Figure 3.10: Tanh activation function: it presents a smooth shape in the output range (-1,1)

Rectified Linear Unit (ReLU)

The ReLU fuction is defined according to the expression:

$$\sigma(z) = max(0, wx + b)$$

Hence, the output will be a classic ramp for positive inputs, 0 otherwise. Although it seems very similar to a linear unit activation function, ReLU presents several advantages. First of all, its non-linearity gives it good approximation properties, differently from the simple linear unit. Moreover, due to its nature, it allows to a restricted part of neurons to fire. In this way the network will be lighter from a computational point of view. It also involves simple mathematical operations with respect to sigmoid like functions. ReLU is probably the most used activation function in deep learning, not only for the already mentioned benefits. It resulted to be an effective solution for more complex issues such as the *vanishing or exploding gradient*.

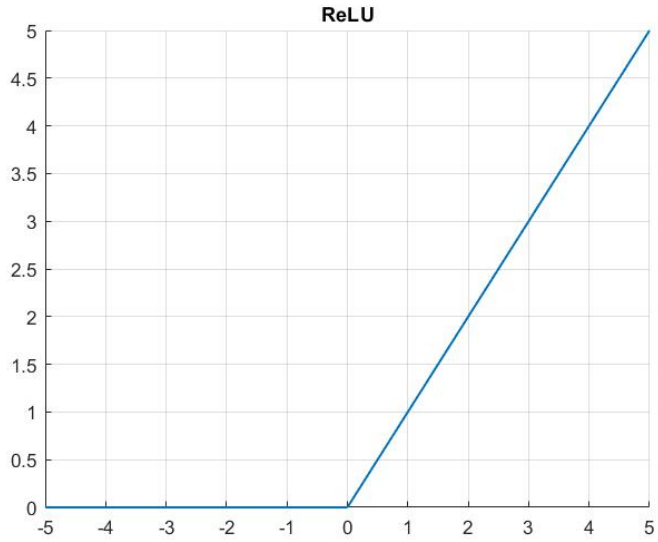

Figure 3.11: ReLU activation functions.

Softmax activation function

The *softmax* function, also known as *normalized exponential function*, is a widely used activation unit. It is especially chosen for the output layer of neural networks, in particular for classification purposes.
The standard softmax expression is:

$$\sigma(z_j) = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

The main property of softmax function is that it can be interpreted as a probability distribution. In other words, the output of the network tells us which is the probability of a sample to be classified with the label of a certain category. This can be clarified by looking at the expression of the function: the exponential of each component of the vector **z** is divided by the sum of all the exponentials.

$$\sum_j \sigma(z_j) = \frac{e^{z_j}}{\sum_k e^{z_k}} = 1$$

It provides a confidence score related to the network's prediction, which is a precious information about its performance.

3.4.5 Learning process: gradient descent

At this point, the architecture of a generic artificial neural network should be clear. It is now possible to explain the main concepts about the learning process of ANNs. Fist of all, the goal is to obtain a collection of weights and bias for the model which provide a correct output, according to the task. The key concept is that we need a measure to evaluate how the network is adapting its weights. For such a purpose, a *cost function* is introduced.

$$C(w,b) = \frac{1}{2n}\sum_x ||y(x) - a||^2$$

In the expression above, w and b are the weights and bias of the network, n the total amount of samples used for the learning process. With x we refer to the single training input, whilst $y(x)$ will be the desired output and a the actual output of the network. This specific form of quadratic cost function is also known as *mean square error* (MSE). Basically, it is a measure of the error committed when the output a is predicted with respect to the desired one $y(x)$. Of course, it is a function of the network's parameters. When $C(w,b)$ is almost 0 for all training inputs it means the training algorithm is working well and a set of suitable w and b have been found. Hence, a minimization problem for $C(w,b)$ has to be solved. Since the number of variables involved in a neural network is huge, an analytical approach would be very difficult to be carried out. An algorithm called *gradient descent* is therefore used. For a simpler initial step, a generic n-dimensional input array v is considered. For small variations of each variables v_j it is possible to express the variation of the cost function in the following way:

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2 + ... + \frac{\partial C}{\partial v_j}\Delta v_j + ... + \frac{\partial C}{\partial v_n}\Delta v_n$$

A more compact form of the expression above can be rewritten by exploiting the concept of gradient of C:

$$\nabla C = \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T$$

$$\Delta C \approx \nabla C \cdot \Delta v$$

where Δv is the vector representation of the variations of v.

The gradient's notation is useful because it directly relates the variations of v with the ones of $C(v)$. At this point, we are interested in finding a set of Δv such that ΔC is negative. The reason behind that can be roughly explained with a visual metaphor. It it sufficient to imagine the cost function as a deep valley. Starting from a random point on its surface, the goal of the process is to reach its bottom

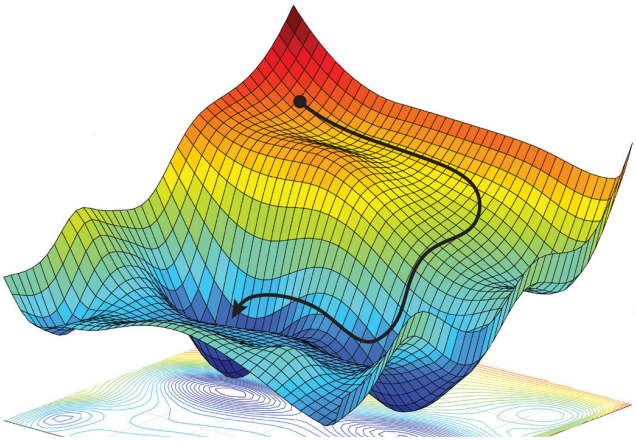

Figure 3.12: Gradient descent visualization on a 3D surface.
[20]

(see Figure 3.12). Hence, it is necessary to choose the appropriate movements Δv to go down correctly, which corresponds to a negative ΔC. This concept can be expressed with the following, considering also a small positive parameter called *learning rate*:

$$\Delta v = v' - v = -\eta \nabla C$$

The representative equation of gradient descent can be obtained by combining the last two equations:

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta ||\nabla C||^2$$

Therefore, an update rule for the parameters v is provided by the algorithm:

$$v \to v' = v - \eta \nabla C$$

To sum up, by choosing a suitable set of changes in the parameters, it is possible to minimize a cost function $C(v)$ with gradient descent. A correct choice of the learning rate is also crucial to tune the process. It has to be small enough to guarantee a good approximation of ΔC especially in the final steps of the algorithm, when the goal is close and fine adjustments are needed (see Figure 3.13). On the other hand, when the global minimum of the cost function is still far, it should not speed down the process too much. For this reasons it is often modified during the process according to an update rule.

Finally, it is possible to express the update rule provided by the gradient descent algorithm using the weights and biases of a neural network. This formulation

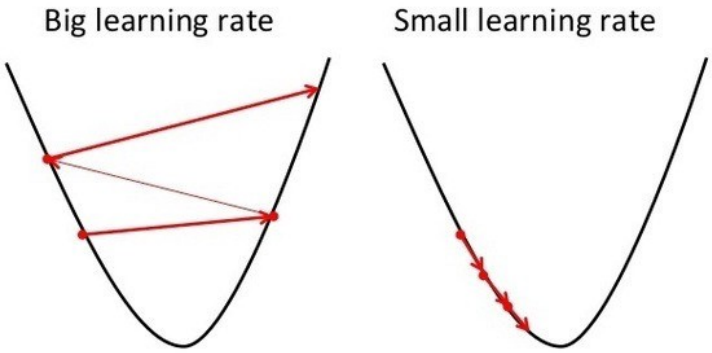

Figure 3.13: A comparison between a big learning rate and a small one when using gradient descent.

describes how ANNs are actually trained. For a j^th weight w_j and bias b_j it looks like

$$w_j \to w'_j = w_j - \eta \frac{\partial C}{\partial w_j}$$

$$b_j \to b'_j = b_j - \eta \frac{\partial C}{\partial b_j}$$

3.4.6 Stochastic Gradient Descent

Beside the working principal of gradient descent, it has to be said that its effectiveness in training neural network can be improved. By looking more in detail the cost function $C = \frac{1}{n} \sum_x C_x$ it is possible to notice that it is an average over all the cost contribution C_x of each training input x, with $C_x = \frac{||y(x)-a||^2}{2}$. This means we need to compute gradients ∇C_x for each training input, resulting in a slow convergence of the algorithm. To speed it up, a modified version is usually chosen. It is known as *stochastic gradient descent* and it consists in considering a limited subset of m out of n samples to compute the gradient ∇C.

$$\nabla C = \frac{1}{n} \sum_x \nabla C_x \approx \frac{1}{m} \sum_j \nabla C_j$$

The small subset of m samples is usually called *mini-batch*. It is possible to express the update rule of weights and biases taking care of this.

$$w_j \to w'_j = w_j - \frac{\eta}{m} \frac{\partial C}{\partial w_j}$$

$$b_j \rightarrow b'_j = b_j - \frac{\eta}{m}\frac{\partial C}{\partial b_j}$$

The stochastic gradient descent selects in a random way a mini-batch from the training data for each iteration of the algorithm. When all have been used once, a *training epoch* is finished and a new cycle is started. The number of epochs required to finish a training process depends on the specific network's size and on the other parameters influencing the process, such as the learning rate and the dimension of the mini-batches. A batch size of 1 can be also chosen. In this extreme circumstance the neural network learns from a single sample at a time. Hence, this case is known as *online* or *incremental learning* and it is very close to human intelligence learning.

3.4.7 The back-propagation algorithm

The gradient descent method allows to find a suitable set of weights and biases. However, the *back-propagation* algorithm still need to be explained in order to better understand how the gradients computation happens.
Today, the back-propagation approach is a pillar of neural network's success. Basically, it consists in computing partial derivatives of the cost function with respect to all the weights of the network. As the name suggests, it works starting from the output to the first layer of the model. Before providing a brief description and comment about the mathematical core of the algorithm, a definition of the error δ_j^l must be given.

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}$$

Where the notation used refers to the j^th neuron in layer l, and z_j^l is the input received by such neuron. In other words, δ_j^l is a variable used to measure the error committed by the neuron.

Back-propagation can be formulated by using four principal equations.

Equation 1: it computes the error at the level of the output layer. It is composed by two terms. The first one tells us the influence of a specific neuron's output a_j^l on the cost function. The second one takes in account the response of the activation function σ when the input z_j^l changes.

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\delta'(z_j^L)$$

The global error can expressed thanks to the matrix-based notation, which involves

the element-wise product.

$$\delta^L = \nabla_a C \odot \sigma'(z^L)$$

Equation 2: it compute the error δ^l in layer l from the error in the next layer δ^{l+1}. This is a fundamental step and the core of back-propagation principle, since it moves the error from a layer to the previous one, taking care of the activation functions in between.

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^L)$$

Equation 3: it express the influence of any bias in the network on the cost function.

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j$$

Equation 4: it express the influence of any weight in the network on the cost function.

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$$

Hence, these four equation describe the essence of the algorithm. It computes the error in the output layer first and then propagates it back until the input layer. Thanks to that, it is possible to compute the partial derivatives $\frac{\partial C}{\partial w^l_{jk}}$ and $\frac{\partial C}{\partial b^l_j}$, which are required for the gradient descent.

For a complete view on the learning process described until this point, a schematic summary of the whole training process is reported below. It shows how a stochastic gradient descent learning algorithm works in combination with back-propagation, with a mini-batch of m training samples for a single training epoch.

1. A mini-batch of m inputs is sampled from the dataset.
2. For each input x in the mini-batch:
   - x enters in the input layer;
   - *Feed-forward*: it passes through all the other layers. For each layer $l = 2,3,...,L$ $z^{x,l} = w^l a^{x,l-1} + b^l$ and $a^{x,l} = \sigma(z^{x,l})$ are computed until the final output.
   - *Output error* $\delta^{x,L}$: compute the error in the output layer according to equation 1 of back-propagation

$$\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L})$$

- *Back-propagate the error:* according to equation 2 of back-propagation, for each $l = L-1, L-2, ...,2$ compute:

$$\delta^{x,l} = ((w^{x,l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l})$$

3. According to the stochastic gradient descent update rule for the weights and biases, for each layer $l = L, L-1, ...,2$ do

$$w^l \to w^l - \frac{\eta}{m} \sum_x \delta^{x,l} \left(a^{x,l-1}\right)^T$$

$$b_j \to b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$$

4. Another mini-batch is sampled from the available dataset until they all have been processed once and the epoch is over.

This is the basic algorithm to train a generic neural network. Further interventions can be thought when the learning process does not work properly. A selected list of possible actions and methods to improve the training of a neural network are briefly discussed in the following section.

3.5 Insights on training neural networks

In this section some important methodologies to improve the training process of neural networks are presented. However, before proceeding, a wider framework on machine learning can be introduced for a better clarification. The architecture described in this chapter is the basic one, also known as *feed-forward* neural networks, since the numerical signal moves in only one direction. Different architectures have been developed along the years, from long short-term memory units to recurrent neural networks. Moreover, the learning process explained in this chapter is called *supervised learning*. It makes use of labeled data to compute the output error and train the network. In practice, many variants of the learning process have been already conceived to tackle the cumbersome procedure of collecting huge labeled datasets. Among them, *unsupervised learning* and *reinforcement learning* are the principal ones.

At this point, it is possible to take in consideration the most common difficulties in training deep neural networks. The collection of techniques and tools reported represents today another building block of deep learning. It is convenient to be familiar with them for a better understanding of the practical implementation of neural networks.

3.5.1 Learning slowdown

A common issue with neural networks is a slow learning process. Without a solid experience in the field, it is often not trivial to fully understand the behaviour that comes out from a network. Here, the *neuron saturation* phenomenon will be briefly described. It is known that neurons learn from the changes in the weights and biases, and the rate of learning is associated with the partial derivatives $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$. As we have already seen a quadratic cost function of the form $C = \frac{(y-a)^2}{2}$ presents the following partial derivatives:

$$\frac{\partial C}{\partial w} = (a - y)\sigma'(z)x$$

$$\frac{\partial C}{\partial b} = (a - y)\sigma'(z)$$

Hence, the speed of learning strongly depends on the first derivative of the activation functions $\sigma'(z)$. By recalling the shape of the sigmoid function (see Figure 3.9) it is clear that $\sigma'(z)$ is close to 0 when $\sigma(z)$ is equal to 0 or 1, i.e. when the function is flat. This is the reason behind learning slowdown.

The cross-entropy cost function

A possible solution to solve the issue of the neuron saturation is to replace the quadratic cost function with the *cross-entropy* cost function. It is defined by the expression:

$$C = -\frac{1}{n}\sum_x \sum_y \left[y_j \ln\left(a_j^L\right) + (1 - y_j)\ln\left(1 - a_j^L\right)\right]$$

The associated partial derivative with respect to the weight for a single neuron is:

$$\frac{\partial C}{\partial w} = \frac{1}{n}\sum_x x_j(\sigma(z) - y)$$

Thus, now the rate of learning is correlated to the quantity $(\sigma(z) - y)$ which is nothing but the output error. This is perfectly reasonable for a learning unit. For the bias, the result is analogue. The cross-entropy cost function is always preferred to the quadratic one whenever sigmoid activation function are used in the output layer.

The log-likelihood cost function

In an analogue way, the *log-likelihood* cost function is used in combination with the softmax unit in the output layer. This also mitigates the learning slowdown.

Given the input x, the expression of the log-likelihood is:

$$C = -\ln(a_x^L)$$

As it can be expected, the partial derivatives with respect to weights and biases results to be directly proportional to the output error.

$$\frac{\partial C}{\partial w_{jk}^L} = a_j^{L-1}(a_j^L - y_j)$$

$$\frac{\partial C}{\partial b_j^L} = (a_j^L - y_j)$$

Weights initialization

The introduction of new cost function works well for the output neurons. However, to avoid the same issue in hidden layers, an alternative solution must be exploited. It turns out that the initial value assigned to weights and biases in the layers plays a fundamental role in the learning process. In particular, a recognized working possibility consists in initializing the weights according to a normal probability distribution with 0 mean and standard deviation equal to $\frac{1}{\sqrt{n_i}}$, where n_i is the number of input connections. This is a quite effective in mitigating the slowdown issue. More advanced methods have been developed by Xavier Glorot and Yoshua Bengio. Today, the result of their studies is known as *Xavier initialization*. According to this theory, the values for weights are picked from a random uniform distribution bounded between $\pm\frac{\sqrt{6}}{\sqrt{n_i+n_{i+1}}}$, where n_i is the number of input connections and n_{i+1} the number of output connections. It turns out that Xavier initialization usually reduced the time for training with respect to standard methods.

3.5.2 Data overfitting

Overfitting is another well known issue affecting neural networks. It occurs when a model is designed to fit extremely well on some data, increasing its level of specificity. It is common to incur in overfitting when using models with a high number of parameters such as neural networks. Overfitting must be strongly avoided in machine learning, since a neural network which provides good results only with data contained in the training set is useless. The ability to generalize the performance on different data can be improved with several methods.

Splitting data

A fist possible method to reduce overfitting is to split the available data in three different subsets. The new *training dataset* will include only a subgroup of the

original amount of data and it will be used for the proper learning process. Then, a *validation dataset* will be exploited to monitor the performance of the network at the end of each epoch. This is a key step, indeed overfitting can be detected by looking at the accuracy gap between the training and the validation set. Finally, the network is tested on a *Test dataset*. A graphical representation of the accuracy reached by the the network along the training process, on both training and validation sets can be useful also for a better tuning of the *hyper-parameters*. An trial and error procedure or a grid-search are often used to look for a suitable combination of learning rate, mini-batch size, number of epochs, learning rate's decay policy. A simple strategy is to stop the training when the accuracy level in the validation set becomes stable.

Data augmentation

The availability of a rich dataset is only guaranteed in machine learning. Hence, training a deep neural network with plenty of parameters results to be difficult. An accurate tuning of the hyper-parameters sometimes is not enough to avoid overfitting. Data augmentation is a particular strategy developed for these purposes. For example, in the specific case of image classification, it is possible to expand the training dataset with artificial samples. A set of different transformation such as rotation, cropping, flipping or filtering can be applied to images in order to artificially create new ones. This is particularly useful when the number of samples for each class in the training set is strongly unbalanced, often because some kind of data is more difficult to collect.

Regularization

Beside data augmentation, there are other chances to reduce overfitting without the necessity of an augmented dataset, for instance exploiting some *regularization* techniques. The two most popular methods are known as *L1 regularization* and *L2 regularization*. The key concept of both methods is to add a 'penalty' or *regularization term* to the cost function used for the learning process. For example, a cross-entropy cost function can be considered.

The L2 regularization, which is probably the most popular, is analysed first. The regularized cost function assumes the following shape

$$C = -\frac{1}{n}\sum_{xy}\left[y_j \ln\left(a_j^L\right) + (1-y_j)\ln\left(1-a_j^L\right)\right] + \frac{\lambda}{2n}\sum_{w} w^2$$

As shown, the regularization term for the L2 method is composed of the sum of the squared weights and of a multiplicative factor. It is responsible of reducing the value of the selected variables. Basically, during the learning process the network

has to choose certain weights such that a good trade-off between the two terms of the new cost function is found. To highlight the concept better, it is possible to rewrite the expression using the notation C_0 to indicate the original cost function.

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2$$

The role of λ, which is a positive *regularization parameter*, is central to make things work. According to its value the relevance of the second term with respect to the first one can be tuned. A small λ makes the regularization term negligible, a large λ increases the importance of learning small weights. Considering a stochastic gradient descent learning algorithm, a new update rule for the weights can be computed with the L2 regularized cost function.

$$w \to w' = w \left(1 - \frac{\eta\lambda}{n}\right) - \eta \frac{\partial C_0}{\partial w}$$

Differently, in the L1 regularization technique the extra term contains the sum of the absolute values of the weights in the networks and the regularized cost function is expressed with

$$C = C_0 + \frac{\lambda}{2n} \sum_w |w|.$$

In an analogue way, the resulting update rule will be:

$$w \to w' = w \frac{\eta\lambda}{n} sgn(w) - \eta \frac{\partial C_0}{\partial w}$$

By comparing the expressions related to L1 and L2 regularization, it is possible to say that both of them aims to penalize large weights in the network. However, in L1 regularization weights are reduced by a constant amount toward 0, whilst in L2 regularization the reduction is proportional to the the weight itself. This means L2 is particularly effective with larger weights. On the contrary the impact of L1 regularization is much bigger when $|w|$ is very small, which are shrink to 0. The result is that it focus the non-null parameters into a selected group of important connections.

From a broader point of view, also the usage of a validation set can be considered a regularizing contribute to the learning process. This is especially true for non-parametric algorithms in machine learning such as k-nearest neighbours, which do not explicitly make use of a cost function.

Dropout

Dropout is a totally different way of acting on the learning process to regularize it. With respect to L1 and L2 regularization, it does not act on the cost function.

In particular, for each training step it randomly selects a certain percentage of neurons contained in a layer and it turn them off (usually around 50%). For these neurons weights and biases will not be updated. Hence, only the remaining active neurons are able to create connections with the neighboring layers. The dropout can be also thought as an averaging process among different neural networks. In some sense, this peculiar approach avoid the network to rely on a restricted number of connections, making its performance more robust.

3.5.3 ADAM optimizer

Beside Stochastic Gradient Descent (SGD), many different algorithm have been developed to solve the optimization problem in the neural networks learning process. Among them, ADAM optimizer algorithm deserves a greater focus for two reasons. Although SGD can be still considered the most popular one, ADAM is increasingly spreading as one of the principal optimizers in Deep Learning. Moreover, it is the actual algorithm used in the practical implementation of this project and it will appear in the following chapters.

ADAM is an adaptive learning rate method. In other words, learning rates are modulated specifically for each different parameter during the update step. This is done by estimating the first and the second moments of the gradient. Before looking at the entire algorithm, it is convenient to introduce the concept of *moment*. It can be defined as the expected value of a random variable to the power of n.

$$m_n = \mathcal{E}[X^n]$$

Hence, the first moment of a random variable is equal to its mean, whilst the second one is the uncentered variance. For the estimation of such quantities for the gradient, ADAM makes use of exponentially moving averages m and v computed with the gradient obtained from the current mini-batch:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Where g is the gradient on the the mini-batch, β_1, β_2 constant hyper-parameters usually fixed at 0.9 and 0.999. These estimators are biased, hence they need a proper correction. The final expression for the estimator results to be:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Therefore, the update rule for the weight during training will be:

$$w_t = w_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Where η is the step size and ϵ is necessary for numerical stability. According to the definition given by Kingma and Ba in the original paper of 2015 [21], ADAM algorithm is reported below.

Algorithm 2 ADAM optimizer algorithm. Popular default values for the constants are $\eta = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$.

Input: stepsize η, ϵ for numerical stability, $\beta_1, \beta_2 \in [0, 1)$ exponential decay rates for the moment estimates, $f(\theta)$ the stochastic objective function, the initial vector of parameters θ_0.

Output: Resulting parameters θ_t.

$m_0 \leftarrow 0$ ▷ First moment estimate vector set to 0

$v_0 \leftarrow 0$ ▷ Second moment estimate vector set to 0

$t \leftarrow 0$ ▷ Timestep set to 0

while θ_t not converged **do**

$t \leftarrow t + 1$

Compute gradient w.r.t parameters θ: $g_t \leftarrow \nabla_\theta f(\theta_{t-1})$

Update of first-moment and second-moment estimates, biased

$$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$

$$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$$

Bias-correction of estimates

$$\hat{m}_t \leftarrow \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t \leftarrow \frac{v_t}{1 - \beta_2^t}$$

Update parameters

$$\theta_t \leftarrow \theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

end while

3.6 Convolutional Neural Networks

In the previous sections, artificial neural networks have been introduced. It must be pointed out that the architecture which has been taken in consideration only involves fully-connected layers. In this case, each neuron inside a hidden layer is connected to all the neurons of the next layer and of the previous one. This architecture is not very efficient when dealing with input data such as images. Principally, this is true for two reasons. First, the spatial structure of the image is not considered. Second, the network becomes too computationally expensive with a huge number of parameters to train. Here *Convolutional Neural Networks* (CNNs) are introduced. The architecture of CNN is particularly optimized for image classifications or other visual based tasks. We will try to give a simple explanation of how does the convolutional operations actually works in CNN and why it is efficient. The three pillars of CNN can be identified in the following concepts:

- *local receptive field*;
- *shared weights*;
- *pooling*

In order to avoid a full connection between input pixels and neurons of a hidden layer, each neuron is associated to a small region of the image, for example a 5x5 square of 25 pixels. This is called the local receptive field of the neuron. It learns a weight for each connection and a general unique bias. Hence, each receptive field region in the input image is connected to a neuron of the first neighboring layer. By sliding the local receptive field by one pixel (or by a general quantity called *stride*), a connection with the second neuron of the hidden layer is created. This operation is repeated until completeness of the input image. The resulting number of neurons in the hidden layer will be:

$$n_h = \frac{W - F - 2P}{S} + 1$$

Where, W is the image width, F is the receptive field size, S is the stride and P is the zero-padding. Sometimes it is useful to set to zero the pixels along the border of the image to have a better control on output size of the layer. For example, for 28x28 input image and a 5x5 receptive field we will have 24x24 neurons in the hidden layer. What is important to say at this point is that all such neurons share the same weights and biases. Indicating with σ a generic activation function and with $a_{x,y}$ the input activation at position x,y, the output of a j,k-th hidden neuron will be:

$$out_{j,k} = \sigma\left(b + \sum_l \sum_m w_{l,m} a_{j+l,k+m}\right)$$

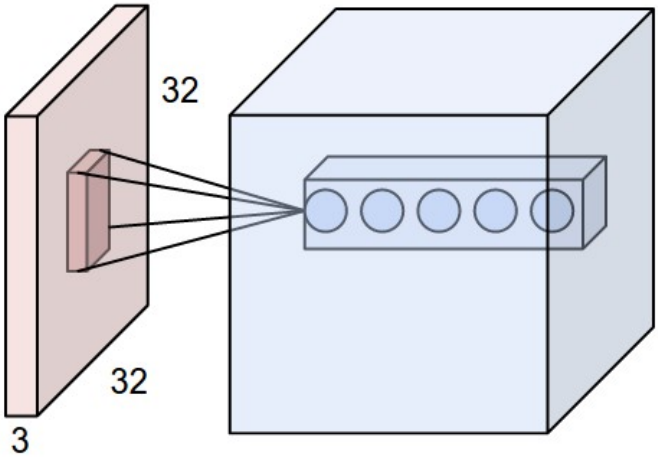

Figure 3.14: An example of input 3 channel image 32x32x3 mapped to a first hidden layer with 5 feature maps.

This assumption implies that a single hidden layer can learn a single feature in the input image, at different locations. This is why the map from input to hidden layer is called *feature map* in CNNs. The shared weights and bias together identify a *kernel* or *filter*. Sharing weights, the number of total parameters in the network is dramatically reduced with respect to a fully-connected based architecture, enabling a faster training process. The number of feature maps is a design parameter for the convolutional layer that depend on the task and on the input image.

Furthermore, a convolutional layer is often associated to a *pooling* layer. Pooling is usually a simple operation performed on small input regions in order to simplify the information contained. A straightforward and popular example is the max-pooling. By considering a 2x2 input region, it outputs the maximum activation. Thanks to this, a further reduction of neurons is achieved.

To sum up, a basic complete architecture of a CNN is composed of convolutional layers, pooling layers and finally fully-connected layer. The data volume is squeezed all along the convolutional section. The training algorithms described before, such as SGD and ADAM, and backpropagation work in the same way for CNN.

Chapter 4

Deep Reinforcement Learning

4.1 Chapter overview

In this chapter the Reinforcement Learning framework is explored, starting from an introduction of the main concepts. Then, tabular RL methods are presented, among them Dynamic Programming, Monte Carlo methods and Temporal-Difference learning are briefly discussed for completeness. Some examples of the principal algorithms are also given. In the last part of the chapter, approximate solution methods are treated with a great focus on Deep Reinforcement Learning. The Deep Deterministic Policy Gradient algorithm, the one used in this thesis, is finally presented. It is convenient for the reader to go through the introduction of the basic reinforcement learning definitions to get a better understanding of what has been used in the project.

4.2 Introduction to Reinforcement Learning

The fundamental idea behind many theories of learning is that we learn thanks to the interaction with our environment. It is easy to confirm this concept by recalling many events of our childhood. When a child is learning how to walk or how to ride a bike, he is involved in a trial and error procedure. According to the action he chooses, the environment gives him back a response, that can be for example the hurting sensation perceived when falling. In order to reach the goal, it is necessary to have a good consciousness of the environment and to be aware of the result of a certain action. This causal connection is effectively learnt through experience.

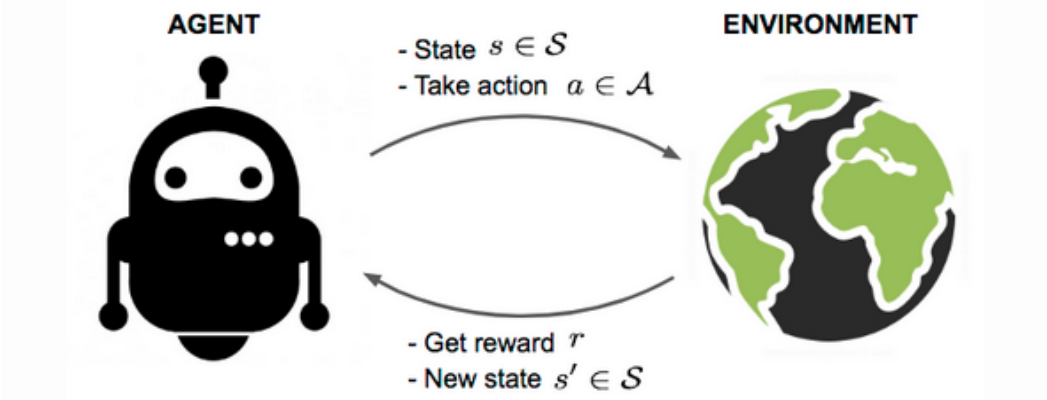

Figure 4.1: The interaction between an agent and the environment typical of reinforcement learning.

[22]

Reinforcement Learning (RL) can be defined as a computational approach focused on learning how to reach a goal by interacting with the environment. More specifically, it is associated to the problem of how to map situations to actions in order to maximize a numerical reward signal. The main peculiar characteristics of reinforcement learning can be identified in the following:

- it involves *closed-loop* problems because the selected action influence the successive inputs.
- the action is chosen by the learner according to a trial and error procedure.
- choosing an action in a certain situation determine the immediate reward as well as the consequent situations and rewards.

For these aspects, it can be said that reinforcement learning is a separate paradigm in machine learning. The differences with supervised and unsupervised learning are quite evident. In reinforcement learning no labeled dataset is exploited and the process focuses on maximizing the reward rather than in finding hidden structure in unlabeled data.

Moreover, a key feature of reinforcement learning is the compromise an agent should respect between *exploitation* and *exploration*. On the one hand, it has to exploit what it has already experienced in order to get reward, on the other hand exploration is important for future action selection.

4.2.1 Elements of Reinforcement Learning

At this point, a more detailed description of reinforcement learning can be discussed. Beside the agent and the environment, it is possible to identify a set of elements that are present in almost every reinforcement learning system: a *policy*, a *reward signal*, a *value function* and a *model* of the environment.

The *policy* of a learning agent defines its way of selecting actions at a given instant of time. It is therefore the core of a reinforcement learning agent, since it is sufficient to give it a certain behaviour. In other words, the policy is responsible of the mapping from a state of the environment to a specific action to perform in that situation. Generally speaking, a policy can be also stochastic, introducing probability in the action selection.

The *reward signal* defines the goodness of a certain event for the agent. At each time step, the environment sends to the agent a numerical evaluation of its behaviour, a *reward*. The agent tries to maximize the total reward over the entire training period. This means the reward signal is responsible of guiding the agent to its goal by indicating good and bad actions.

A *value function* can be roughly defined as a long-term advisor for the agent. It associates to a state a *value* which is correlated to the total reward which is possible to gain in the future starting from that state. Hence, if rewards give the agent an indication of what is good to do in an immediate sense, values take care of the potential development of taking a decision in a certain situation. This is certainly a fundamental role in a reinforcement learning system. For example, an agent can decide to move into a new state gaining a low immediate reward. Nevertheless, this can still be a good choice if it gives the agent the chance to reach next states that yield high rewards.

In any case rewards are considered primary, since values are predictions of rewards and they could not exist without them. However, when selecting an action, we should consider the one that allow to reach states with the highest value, not highest reward, because the total reward accumulated over the long run will be much greater. Unfortunately, an efficient estimation of the value function is not trivial. For this reason this task is a crucial component of almost every reinforcement learning algorithm.

Finally, some reinforcement learning system exploit a *model* of the environment. If this is the case, they are identified as *model-based* methods. A model can be defined as a virtual twin of the environment and it can be useful to make some inference about its future behaviour for planning purposes. On the contrary, *model-free* methods do not make use of any model and they are based on a pure trial and error learning.

4.3 Markov Decision Process

Reinforcement learning aims to frame the problem of goal-based learning from interaction. The *Markov decision process* (MDP) is a way to formally define such a problem. A better clarification of this concept is given in this section trying to put together all the ideas discussed following a step-by-step approach.
As already introduced in the previous section, reinforcement learning is based on the interaction between:

- the *agent*: the learner and decision-maker;
- the *environment*: what is outside the agent and interact with it.

This continuous interaction can be formalized in a closed-loop dynamics as shown in Figure 4.2. At each discrete time step $t = 0,1,2,3...$ the agent receives in input

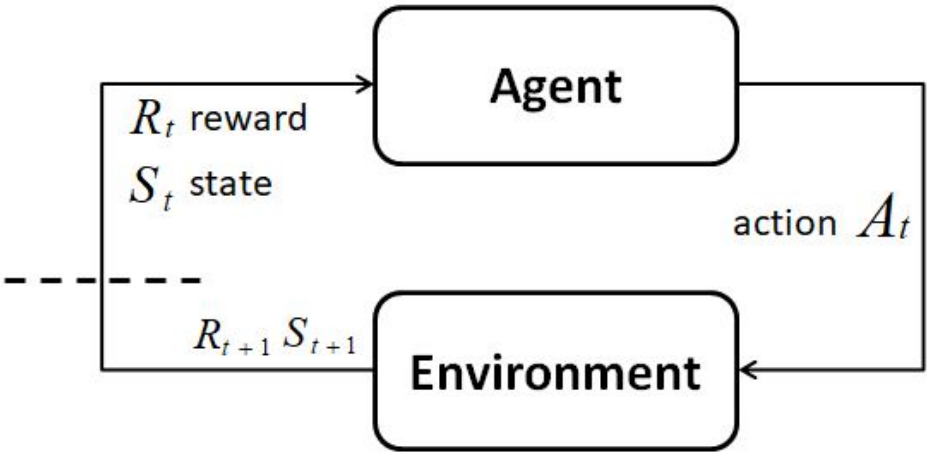

Figure 4.2: Markov Decision Process schematic.

the *state* of the environment $S_t \in \mathcal{S}$, which should contain all relevant information about the environment. The agent chooses an action $A_t \in \mathcal{A}$ based on that. At the next time step the environment sends back a new state S_{t+1} and a reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$. Therefore, the process generates an alternated sequence of signals exchanged between the agent and the environment as the following:

$$(S_0), (A_0), (R_1, S_1), (A_1), (R_2, S_2), (A_2), ...$$

A graphical representation can be useful to better understand a generic set of transitions. An example is reported in Figure 4.3, where circles contains the states and arrows represent the transition from a state to another based on the selected action.

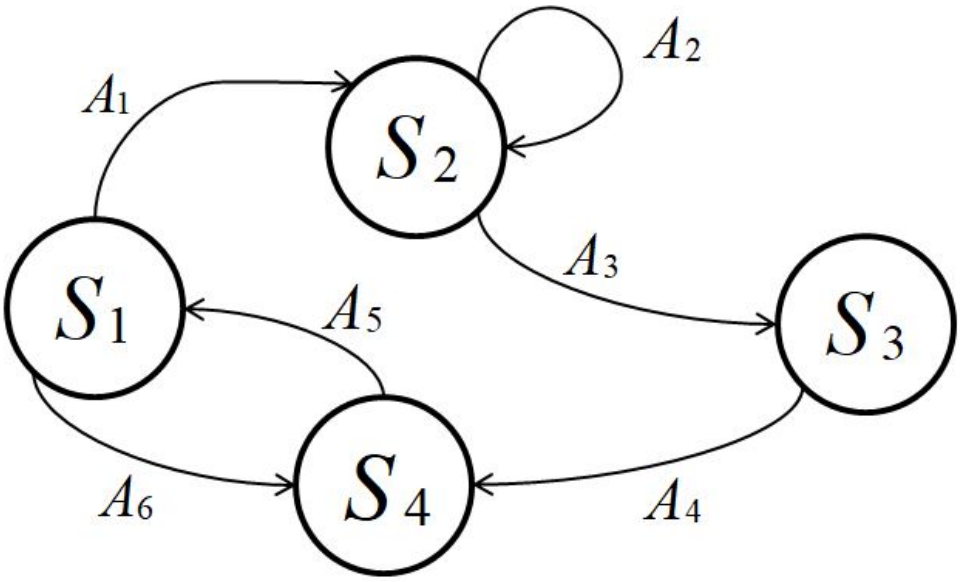

Figure 4.3: Finite Markov Decision Process: a simplified transition schematic.

Now, a generic response of the environment at time $t+1$, also called one-step dynamics, can be expressed by a probability distribution that takes in consideration all the previous events:

$$P_r\{R_{t+1} = r, S_{t+1} = s' | S_0, A_0, R_1, ..., S_{t-1}, A_{t-1}, R_t, S_t, A_t\}$$

However, if the state contains all the relevant information about past transitions the environment's one-step dynamics at time $t+1$ depends only on the state and the action at time t. When this is true, the state signal has the *Markov property*. In this case the previous expression becomes:

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

Moreover, the Markov property can be extended to the whole environment if the previous assumption holds for every s', r. This property has a tremendous relevance for the whole RL conceptual framework. In fact, it means that given the actual state and action, one is able to predict all future states and possible rewards. Markov property is therefore extremely advantageous in reinforcement learning to efficiently choose good actions.

Hence, it is possible to refer to a reinforcement learning task as a Markov decision process whenever the Markov property is satisfied. In the case of finite state and action spaces, the process is called a *finite Markov decision process*.

Goals, Reward and Returns

As already introduced, a reinforcement learning agent has the final goal of maximizing a numerical signal called reward. At each time, the environment assign a reward to the agent. According to the *"reward hypothesis"* [13]:

> That all of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward).

This peculiar idea of goal, strictly related to the reward signal, is typical of reinforcement learning and it is not a limiting formulation for real applications. For example, a robot can learn how to collect empty bottles for recycling simply by assigning a +1 for each bottle collected and a -1 every time it collect a wrong object or miss a bottle. A wide variety of rewards can be thought according to the specific application.

The final amount of reward obtained can be formally called *return* and it can be indicated as G_t. The mathematical expression of G_t depends on the specific problem to tackle, a basic case could be the sum of all the rewards:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T$$

It is possible to notice that a final time instant T concludes the sequence of rewards. This can be the case of agent-environment interactions that happen in separated subsequences called *episodes*. In some sense this episodic interaction can be compared to games levels. Differently, there exist active processes that need a continuous agent-environment interaction to be learnt. In this scenario, it would be $T = inf$ and the return could diverge. A *discount factor* is therefore introduced and the expected discounted return will be:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...+ = \sum_{k=0}^{\inf} \gamma^k R_{t+k+1},$$

where the parameter γ is $0 \leq \gamma \leq 1$ and it is called the *discount rate*. With $\gamma = 1$ the result is unchanged with respect to the previous expression. When instead $\gamma < 1$ the infinite sum will converge to a finite value, given all bounded reward contributions. A peculiar case called *myopic agent* occurs with $\gamma = 0$, since the only immediate reward R_{t+1} is maximized.

Optimal Value Functions and Policies

A formal definition of *value functions* can be given at this point. In particular, it possible to define different value functions for the reinforcement learning framework.

A *state-value function* express how good is a certain state for the agent, according to the associated expected return. In an analogue logic, a *state-action pair value function* can be defined. This last function specifies how good an action is for the state in account.
The concept of *policy* can be formulated as the function that associate the states to the probabilities of selecting the possible actions. According to this definition, an agent which follows a policy π has the probability $\pi(a|s)$ to choose the action a in the state s. The value function under the policy π is indicated with $v_\pi(s)$. It expresses the expected return that the agent should get when starting from state s having a policy π. In a MDP this can be written as:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\inf} \gamma^k R_{t+k+1} \middle| S_t = s\right]$$

where $\mathbb{E}_\pi[\cdot]$ is the expected value of a random variable given the policy π of the agent. v_π is the *state-value function for policy* π.
In an analogue way the *action-value function for policy* π, q_π can be defined:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\inf} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right]$$

Hence, $q_\pi(s,a)$ formally expresses the value of choosing the action a in state s under the policy π.

At this point it is easy to give a definition of optimal policy and optimal value function. It can be said that a policy π is better than another policy π' if its expected return is the greatest among the two of them, for all states:

$$\pi > \pi' \iff v_\pi(s) > v_{\pi'}(s), \forall s \in \mathcal{S}$$

A policy is said to be *optimal* if it is better than or equal to all other policies. An optimal policy is usually denoted as π_*. There could be more than one optimal policy, but all of them will share the *optimal state-value function*, v_*:

$$v_*(s) = \max_\pi v_\pi(s), \forall s \in \mathcal{S},$$

as well as a *optimal action-value function*, q_*:

$$q_*(s,a) = \max_\pi v_\pi(s,a), \forall s \in \mathcal{S},$$

Optimal policies and value functions cannot be found in non-finite MDPs due to practical constraints in the implementation (such as the amount of available memory), however useful approximations can be used.

4.4 Tabular methods for reinforcement learning

In this section, a brief discussion about the simplest reinforcement learning approaches is carried out. These include tabular cases or methods applied with only finite MDPs. Firstly Dynamic Programming is considered only to give the reader a complete point of view about RL. Then, Monte Carlo methods are shortly introduced. Finally, the key concept of Temporal-Difference Learning will be explained.

4.4.1 Dynamic programming

Dynamic programming (DP) consists in a variety of algorithms used to compute optimal policies. Usually, they can be implemented only when a perfect model of the environment is given. Hence, a theoretical case such as an MDP can be one of the few possible case of application. In real scenarios, a perfect environment cannot be found and DP may result to be inappropriate or too computationally expensive. Nonetheless, DP can be successfully used in the financial field.

Iterative policy evaluation is one of the possible method in DP literature. It allows to obtain the state-value function v_π given an arbitrary policy π. This algorithm exploits different results that can be obtained combining the equations described in the previous section. Among them, the starting point for policy evaluation is the *Bellman equation*, here used for computing v_π:

$$v_\pi = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')], \ \forall s \in \mathcal{S}.$$

In this equation, the term $\pi(a|s)$ is the probability of choosing an action a in state s under the policy π. For $\gamma < 1$ the existence and uniqueness of v_π are guaranteed. However, an iterative computational procedure is more appropriate for the purposes of reinforcement learning. Hence, by considering an arbitrary initial v_0, the state-value function can be approximated through successive computational steps using the Bellman equation as an update rule:

$$\begin{aligned} v_{k+1}(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1})|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_k(s')\right], \ \forall s \in \mathcal{S} \end{aligned}$$

Once a value function v_π has been computed, looking for an optimal policy can be a good advancement, since an arbitrary one has been used to compute v_π. The result is a new *greedy* policy called π' that can be obtained combining equations already seen. Here is directly reported without discussing the whole derivation. It can be computed with:

$$\pi'(s) = argmax_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')]$$

The greedy policy takes the action that maximizes the action-value function in the short-term. No further details are provided about further possible optimization of π and v_π since the result is out of the scope of this thesis. A deeper analysis of DP ca be found at [13].

4.4.2 Monte Carlo Methods

With respect to DP algorithm, Monte Carlo methods are not strictly dependent on the environment's knowledge. In fact, they only requires to know the transitions composed of states, actions and rewards obtained through the interaction with a real or a simulated environment. These samples are also called *experience*. In particular, learning from a *simulated* experience is a powerful tool, and it is what is actually done in this thesis project. A virtual model of an environment is required to simulate the interaction and obtain the sample transitions, but the associated probability distributions are not requested as in DP.
Here only episodic tasks are considered for Monte Carlo methods. This allow to have well-defined returns that can be easily averaged over all the episodes of the task. In fact, the approach used for reinforcement learning problem consists in sampling and averaging returns associated to state-actions pairs.
As first goal, we always aim to estimate the value function of a state s under the policy π, $v_\pi(s)$. A particular state can occur several times inside the same episode. The term *visit* is usually used to refer to the occurrence of the state. Based on this definition, it is possible to identify two main Monte Carlo methods:

- The *first-visit MC method* estimates $v_\pi(s)$ averaging the returns coming after the first visit to s.
- The *every-visit MC method* takes in account the returns following all visits to s for the average.

Both the MC algorithms converge to the value function. Here, the pseudo-code of first-visit MC method is reported. The every-visit version is basically the same except for the check of state S_T. Monte Carlo methods allows also to estimate the state-action pair value function $q(s, a)$. More in detail, this is particularly useful when a model of the environment is not known. In this case the state values are not sufficient to determine the actions associated with highest rewards. It is possible to talk about a *visit* of a state-action pair when the agent takes the action a when it is in the state s. Monte Carlo methods illustrated before works in the very same way and are able to estimate the expected values with an increasing number of visits. However, the difference is that with a deterministic policy π, many state-action pairs can never be visited. This means that the agent will not choose among all the possible actions associated to a state, which is the basic purpose of learning action values. In other words, we need to keep a sufficient rate of *exploration*. A

Algorithm 3 First-visit Monte Carlo method for the estimation of $V \approx v_\pi(s)$

Input: a policy π, a positive number of episodes n_e
Output: the estimated value function V

Initialization of returns $R(s) = 0, \forall s \in \mathcal{S}$
Initialization of $N(s) = 0, \forall s \in \mathcal{S}$
▷ N is a counter of the number of visits to each state s
for *episodes* in n_e **do**
    Generate the sequence according to π: $S_0, A_0, R_1, S_1, A_1, R_2, .., S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    for each time step of the episode $t = T-1, T-2, ...,0$ **do**
        $G \leftarrow G + R_{t+1}$
        if *state* S_t *is* ***not*** *present in the sequence* $S_0, ..., S_{t-1}$ **then**
            $R(S_t) \leftarrow R(S_t) + G_t$
            $N(S_t) \leftarrow N(S_t) + 1$
        end if
    end for
end for
$V(s) \leftarrow \frac{R(s)}{N(s)}, \forall s \in \mathcal{S}$

possible approach for this problem is called *exploring starts*. It consists in starting the episode from a specific state-action pair and then assign a non-null probability to each possible action. It guarantees a complete exploration of state-action pairs in an infinite number of episodes but it could be practically not feasible. The most popular alternative approach is to adopt a stochastic policy.

At this point it is possible to briefly describe how Monte Carlo methods are able to approximate optimal policy for control purposes. The main idea of the procedure is very similar to DP, i.e. the *generalized policy iteration* (GPI) is followed. As graphically shown in Figure 4.4 the iterative process consists in updating an approximate value function for the current policy, and the policy is modified at each step according to the value function. This adversarial behaviour results in an approximate optimal policy. As seen before, with an action-value function no models are needed and it is possible to make the greedy policy. Policy improvement constructs each policy π_{k+1} as the greedy policy based on q_{π_k}. This is a direct application of the *policy improvement theorem*, which state:

$$q_{\pi_k}(s, \pi_{k+1}(s)) \geq v_{\pi_k}(s)$$

Basically, whenever a better policy is found by considering its future returns through the value function, the actions start to be chosen according to π_{k+1} instead of following π_k. Until here, two basic assumptions have been considered for policy

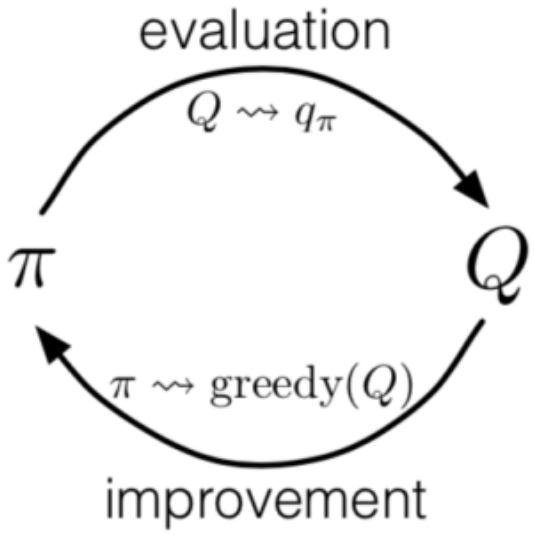

Figure 4.4: Policy improvement scheme.

improvements: an infinite number of episodes for the policy evaluation and the usage of exploring starts. For a practical implementation of the method, these have to be removed. It is easy to pick a sufficient amount of steps to guarantee a good approximation, within certain bounds. When the exploring start is not feasible, it is possible to use particular greedy policy called $\epsilon - greedy$ policy:

$$\pi(a|s) \geq \frac{\epsilon}{|\mathcal{A}(s)|}$$

This means that an action is usually picked up such that it maximizes the action-value in a deterministic fashion. However, with a probability ϵ a random action is selected.

It is now convenient to introduce the concept of *on-policy* and *off-policy* methods. Basically, on-policy methods aims to maintain and improve the policy used to make decisions. On the contrary, off-policy methods tries to improve a different policy. This difference will be useful to understand the following sections. The main idea behind on-policy Monte Carlo methods are based on GPI. Differently, off-policy methods focus on the usage of two different policies. One that we want to become the optimal policy and it is called the *target policy*. The other one, will be devoted to exploration and it is called *behaviour policy*. For example, considering the prediction problem with fixed target policy π and behaviour policy b, the principal aim will always be the estimation of a value function v_π or q_π. If we want to use b to estimate values for pi, what is useful to do is to apply the *coverage*

assumption. In brief, every action taken following π should also be taken under b. For this reason, what is often convenient to do is to use a behaviour policy stochastic in states, for example an $\epsilon - greedy$ policy, not identical to the target one, which instead will be deterministic. Off-policies usually rely on the principal of *importance sampling*. It assign a weight to returns according to their probability to occur. This is used in MC methods when averaging returns from policy b. Without this intervention, they do not lead to the value of the target policy v_π. Off-policies and importance sampling will be treated more in detail in the following section.

4.4.3 Temporal-Difference Learning

Temporal-difference learning (TD) can be considered the central innovative idea behind reinforcement learning. TD presents some elements of both Monte Carlo and Dynamic Programming methods. Indeed TD methods are able to learn without the need of a model of the environment in the same way of MC. Similarly to DP, they bootstrap, i.e. they update estimates using also other learned results. As done for the previously described method, we start introducing the problem of prediction, briefly illustrating how TD estimates the value function v_π for a policy π. If a basic every-visit MC use the obtained return of the visit as target for $V(S_t)$, according to

$$V(S_t) \leftarrow V(S_t) + \alpha \left[G_t - V(S_t)\right],$$

where G_t is the actual return and α is a constant. Differently, TD simplest method makes the update waiting only for the next time step using the obtained reward R_{t+1} and the estimate $V(S_{t+1})$:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right]$$

This method is called TD(0) or also *one-step* TD. The quantity $R_{t+1} + \gamma V(S_{t+1})$ represents the target for TD. Moreover, it is possible to define the TD *error* as follows:

$$\delta = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

The advantages of TD with respect to DP and MC are easy to be noticed. The combination of bootstrapping and independence from a model are precious characteristics especially with long episodes. In addition, TD method do not suffer of typical issues that can arise in MC-based problem such as the discount of episodes used to test experimental action. A good and fast convergence is usually guaranteed with TD in good situations, making these methods convenient in the majority of the situations. Here, the main TD method are introduced and briefly discussed.

SARSA: on-policy TD method

SARSA algorithm takes its name from the quintuple composed by the typical sequence $S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}$. Being an on-policy control method, in SARSA the action-value function $q_\pi(s,a)$ is continually estimated for the behaviour policy π, pushing it toward greediness. The assumption of an infinite number of visits for all state-action pairs and ensures the convergence of SARSA algorithm. The policy can be set to an $\epsilon - greedy$ policy as well as a $\epsilon - soft$ policy. The pseudo-code of SARSA algorithm is shown below.

Algorithm 4 SARSA algorithm for estimating $Q \approx q_*$

Input: a small $\epsilon > 0$ for ϵ-greedy policy π, step size $\alpha \in (0,1]$
Output: the estimated action-value function Q

Initialize arbitrarily $Q(s,a), \forall s \in \mathcal{S}^+, a \in \mathcal{A}(s)$,except $Q(terminalstate,) = 0$
for *each episode* **do**
    Initialize S
    Choose A from S using ϵ-greedy policy
    for each time step of the episode **do**
        Take action A, observe R, S'
        Choose A' from S' using ϵ-greedy policy
        $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma Q(S',A') - Q(S,A)\right]$
        $S \leftarrow S'$; $A \leftarrow A'$;
        if S is terminal **then**
            break loop
        end if
    end for
end for

Q-Learning: off-policy TD method

Q-learning is an off-policy TD control method. It can be considered a real innovation in reinforcement learning. In Q-Learning, the optimal action-value function q_* is approximated by directly learning Q according to:

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1},a) - Q(S_t,A_t)\right]$$

In such a way, the algorithm results to be immediate, as shown in pseudo-code below.

Algorithm 5 Q-learning algorithm for estimating $\pi \approx \pi_*$

Input: a small $\epsilon > 0$ for ϵ-greedy policy π, step size $\alpha \in (0,1]$
Output: the estimated action-value function Q

Initialize arbitrarily $Q(s,a), \forall s \in \mathcal{S}^+, a \in \mathcal{A}(s)$,except $Q(terminalstate,) = 0$
for *each episode* **do**
    Initialize S
    for each time step of the episode **do**
        Choose A from S using ϵ-greedy policy
        Take action A, observe R, S'
        Choose A' from S' using ϵ-greedy policy
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$
        $S \leftarrow S'$;
        if S is terminal **then**
            break loop
        end if
    end for
end for

4.5 Approximate solution method: Deep Reinforcement Learning

In the previous sections, the general framework of RL has been introduced. Successively, tabular methods such as DP, Monte Carlo, TD have been briefly described. It is now possible to move towards the Deep Reinforcement Learning framework. First of all, what it is needed to be considered for this further step is that in reinforcement learning tasks state spaces are frequently huge. Hence, tabular methods present strong limitations due to the cost of updating accurately tables with data in the required time. In this scenario an optimal policy and an optimal value function cannot be found and approximate solutions must be considered according to the available computational resources. What usually happens is that many encountered states will be totally new, and the algorithm should be able to *generalize* the knowledge that has already learnt in order to make sensible decisions. In practice, it has to learn how to behave correctly on a wide number of situations, experiencing a much smaller subset. The generalization challenge is often translated in a *function approximation* problem. In our case, the desired function to approximate is usually a value function.
Although several methods exist for function approximation, for the purpose of this thesis only solutions based on artificial neural networks will be explained. An essential difference in this new framework is that value functions are no more

represented using tables. Instead, they are shaped with a parametric functional form. For ANNs the parameters coincide with the weights of the network $\mathbf{w}(\mathbf{w} \in \mathbb{R}^d)$. Since both the state-value and the action-value functions are now dependent on the vector $\mathbf{w}$, they can be written as $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$ and $\hat{q}(s, a, \mathbf{w}) \approx q_*(s, a)$.

4.5.1 Experience Replay

A popular practice in reinforcement learning is the method of *experience replay*. It has been initially studied by Lin in 1992. However, its recent success is mainly related to its application in the DQN algorithm proposed by Mnih et al. [23] (2013) to learn playing ATARI games with DRL. The DQN algorithm will be described later, here we briefly focus on experience replay. The method is based on saving in a memory buffer called *replay memory*, at each time step, the tuple $(S_t, A_t, R_{t+1}, S_{t+1})$. It contains all the information about the transition of the agent from a state S_t to the next one, choosing a certain action A_t and receiving a reward R_t. Once the replay memory reaches a sufficient number of stored transitions, mini-batches can be sampled uniformly at random. Hence, experience is directly used to train the ANN of the agent. Experience replay provides several advantages. First of all, it increases the data efficiency of the algorithm, since an experienced event can be used to update the agent's weights multiple times. Another fundamental effect of experience replay is to remove the instability in the learning process caused by temporally correlated training samples. Consecutive samples should be always avoided in reinforcement learning. A third improvement that can be done is correlated to the dependence of the target function on the weights. When learning on-policy with a parametric function approximation, parameters determine future samples, which are then used to update the same parameters. This is a source of instability and it is mitigated with experience replay thanks to the smoother learning achieved through averaging the behaviour distribution over a great amount of previous states. Experience replay clearly fits well with an off-policy learning.

4.5.2 Target Network

In addition to experience replay, a second approach can be used to reduce the instability of the algorithm. In particular, Mnih et al. suggested to use another network called *target network* to break the dependence of the target function on weights $\mathbf{w}$. After a certain amount of training steps, the network's parameters can be used to update the duplicate target network, which is used as reference. This improves convergence of the algorithm when using TD-error.

4.5.3 Actor-Critic architecture

A key concept for this thesis is the actor-critic architecture. Differently from the classic MDP agent, this DRL framework involves the usage of two separate entities. The **actor** is responsible of selecting the action, hence it represents the function approximator of the policy. The **critic** evaluates the goodness of what the actor has decided to do. For this reason, it usually approximates an action-value function $Q(s, a, \mathbf{w})$ using the TD error. The critic loss is therefore based on the TD error, whilst the actor network will be updated according to Policy Gradient algorithm, i.e. exploiting the gradient obtained from the critic. Roughly speaking, the critic tries to teach the actor what are good or bad choices. A schematic is reported in Figure 4.5 for a better visualization. During the training process, both the actor and critic networks are updated. However, the actor will be the only entity employed in the desired task, whilst the presence of the critic is limited to the training phase.

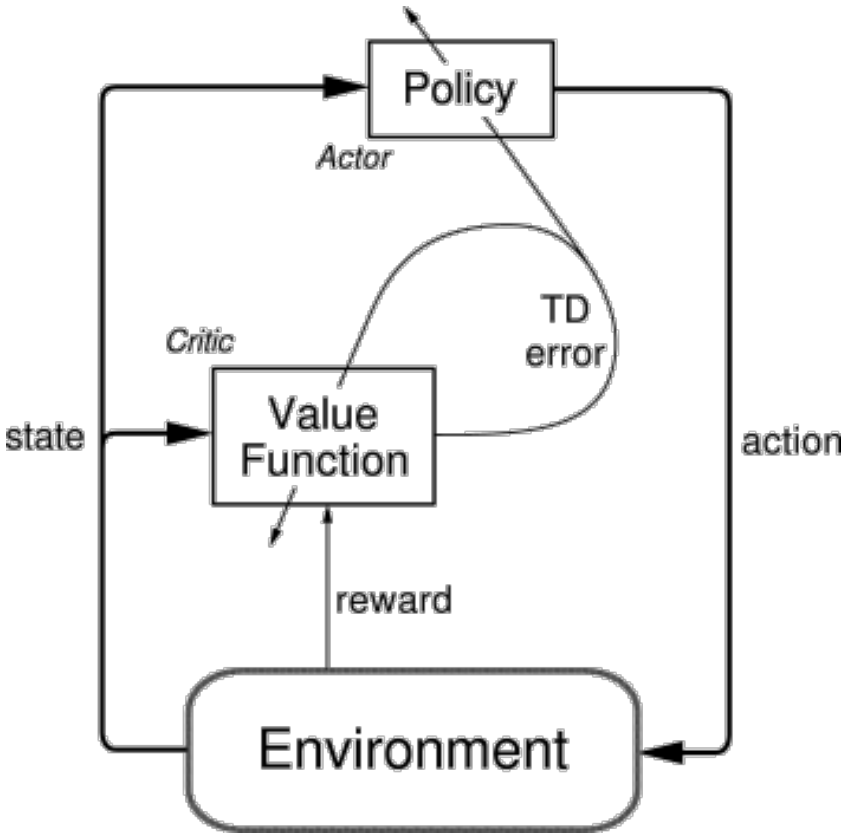

Figure 4.5: Actor-Critic architecture scheme.

4.5.4 Deep Q-Learning algorithm

Deep Q-Learning algorithm is an advanced version of Q-learning method. Similarly, this method focuses on the approximation of the optimal action-value function $Q^*(s, a)$. The definition of optimal action-value function remains the same:

$$Q^*(s, a) = \max_{\pi} \mathbb{E}[R_t | s_t = s, a_t = a, \pi]$$

As explained in the previous sections, it can be exactly computed thanks to the Bellman equation for action-value function:

$$Q^*(s, a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} Q^*(s', a') | s, a\right]$$

Practically, an ANN is used as non-linear function approximator to obtain $Q(s, a; \theta) \approx Q^*(s, a)$, where θ is used to indicate the weights of the network. In this particular case, the neural network is often called *Q-networks* or *Deep Q-network* (DQN), due to its purpose. The Q-network is usually trained with stochastic gradient descent, which aims to minimize the loss function $L_i(\theta_i)$ at each time step:

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(s,a)}\left[\left(y_i - Q(s, a, \theta_i)\right)^2\right],$$

where y_i is the target on iteration i and $\rho(s, a)$ is the probability distribution over states and actions also called *behaviour distribution*. The difference between target policy and behaviour policy has been explained in the previous sections. Here, the behaviour distribution is used to guarantee a sufficient exploration for the agent. The pseudo-code of the algorithm is reported below.

Deep Q-learning is a **model-free** algorithm, since it does not need to know the probability distribution of the environment dynamics. Moreover, it is an **off-policy** method. Indeed it uses a target greedy policy for the optimization of the action-value function, together with an ϵ-greedy policy for the behaviour distribution $\rho(s, a)$. It also exploits the experience replay method, training the networks with mini-batches sampled from the a total of N transitions stored in the memory buffer D.

Algorithm 6 Deep Q-learning algorithm with experience replay

Input: a small $\epsilon > 0$ for exploratory ϵ-greedy policy, replay memory D of capacity N, number of episodes M.

Output: the function approximator of the action-value function Q

Initialize the replay memory D with capacity N.
Initialize the Q function approximator with random weights.
for episode = 1, M **do**
    Initialize s
    for each time step $t = 1, T$ **do**
        Generate a random number $0 \leq h \leq 1$
        if $h \leq \epsilon$ **then**
            Pick a random action a_t
        else
            Select $a_t = \max_a Q^*(s, a; \theta)$
        end if
        Perform selected action in the simulated environment and observe r_t, s_{t+1}
        Set next state $s_{t+1} = s_t$
        Store experience transition (s_t, a_t, r_t, s_{t+1}) in replay memory D
        Sample mini-batch of transitions (s_j, a_j, r_j, s_{j+1}) from replay memory D
        Set target: $y_j = \begin{cases} r_j \textit{ if final state} \\ r_j + \gamma \max'_a Q(s_{j+1}, a'; \theta) \textit{ otherwise} \end{cases}$
        Perform a gradient descent step on loss $(y_i - Q(s, a, \theta_i))^2$
    end for
end for

4.5.5 DDPG Algorithm

Deep Deterministic Policy Gradient is the last algorithm described in this chapter. Indeed it is the actual DRL algorithm used in the implementation of this thesis project. It is now possible to define and to frame DDPG according to the entire set of definitions and methods seen before. First of all, DDPG is a **model-free off-policy actor-critic** algorithm. Hence, there is no need to know the environment dynamics and, as already seen for Deep Q-Learning, a target deterministic policy is used in combination with a stochastic exploratory policy. Similarly to DQN, experience replay and target networks are used also in DDPG. From a broader perspective, DDPG can be considered an extension of DQN from discrete to continuous space. The actor-critic architecture is also a substantial difference.

Before proceeding with the illustration of the algorithm, originally proposed by Lillicrap et al. in 2015 [24], Deterministic Policy Gradient (DPG) methods are

briefly introduced. Among all the existing versions, here we are interested in DPG methods for off-policy learning with actor-critic architecture. Policy gradients methods exploits gradient ascent technique to optimize the policy performance objective function, generally defined as

$$J(\pi) = \mathbb{E}[r_1^\gamma | \pi]$$

It expresses the cumulative discounted reward, which has to be maximized. When dealing with off-policy learning in continuous space, it has to be modified such that it becomes the value function of the target policy, averaged over the state distribution of the behaviour policy [25]:

$$\begin{aligned} J_\beta(\mu_\theta) &= \int_{\mathcal{S}} \rho^\beta(s) V^\mu(s) ds \\ &= \int_{\mathcal{S}} \rho^\beta(s) Q^\mu(s, \mu_\theta(s)) ds \end{aligned}$$

Where μ_θ is the deterministic policy dependent on the weights θ. β instead is the behaviour policy used to sample the transition guaranteeing exploration of the unknown environment. The off-policy deterministic policy gradient can be computed as:

$$\begin{aligned} \nabla_\theta J_\beta(\mu_\theta) &\approx \int_{\mathcal{S}} \rho^\beta(s) \nabla_\theta \mu_\theta(a|s) Q^\mu(s,a), ds \\ &= \mathbb{E}_{s\sim\rho^\beta}\left[\nabla_\theta \mu_\theta(s) \nabla_a Q^\mu(s,a)|_{a=\mu_\theta(s)}\right] \end{aligned}$$

The action-value function is approximated by the critic $Q^w(s,a) \approx Q^\mu(s,a)$. The update rules for the DPG are:

$$\delta_t = r_t + \gamma Q^w(s_{t+1}, \mu_\theta(s_{t+1}) - Q^w(s_t, a_t)$$

$$w_{t+1} = w_t + \alpha_w \delta_t \nabla_w Q^w(s_t, a_t)$$

$$\theta_{t+1} = \theta_t + \alpha_\theta \nabla_\theta \mu_\theta(s_t) \nabla_a Q^w(s_t, a_t)|_{a=\mu_\theta(s)}$$

At this point, it is possible to talk about DDPG [24], introducing ANNs as function approximators. The loss function is defined, similarly to Deep Q-learning:

$$L(\theta^Q) = \mathbb{E}_{s_t\sim\rho^\beta, a_t\sim\beta, r_t\sim E}\left[(Q(s_t, a_t|\theta^Q) - y_t)^2\right]$$

where y_t is the target,

$$y_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1})|\theta^Q)$$

Experience replay and target networks update are used in DDPG. More in detail, to improve the convergence of the algorithm, a *soft* update is performed. By denoting the target copies of both actor and critic as $Q'(s, a|\theta^{Q'})$ and $\mu'(s|\theta^{\mu'})$, the soft update is performed as follows for the two of them:

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

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

With τ a small positive parameter. Moreover, in the original implementation of the paper by Lillicrap et al., an exploration policy is constructed by adding to the actor policy a noise $\mathcal{N}$, generated with a particular process (Ornstein-Uhlenbeck). A detailed explanation is avoided here, the resulting expression of the policy is shown below.

$$\mu'(s_t) = \mu(s_t|\theta_t^{\mu}) + \mathcal{N}$$

However, in this thesis a basic stochastic policy is used to guarantee a non-null probability of selecting random actions for exploration. The pseudo-code of the used DDPG algorithm is shown in the next page.

Algorithm 7 DDPG algorithm

Input: a small $\epsilon > 0$ for exploration, replay memory R of capacity N, number of episodes M, mini-batch size m.

Output: the critic network (function approximator of the action-value function Q), the actor network (approximator of the target policy μ).

Initialize the replay memory R with capacity N.
Randomly initialize the critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$
for episode = 1, M **do**
    Initialize observation s_1
    for each time step $t = 1, T$ **do**
        Generate a random number $0 \leq h \leq 1$
        if $h \leq \epsilon$ **then**
            Pick a random action a_t
        else
            Select $a_t = \mu(s_t, \theta^\mu)$
        end if
        Perform selected action in the simulated environment and observe r_t, s_{t+1}
        Set next state $s_{t+1} = s_t$
        Store experience transition (s_t, a_t, r_t, s_{t+1}) in replay memory R
        Sample a random mini-batch of m transitions (s_i, a_i, r_i, s_{i+1}) from R
        Set target: $y_j = \begin{cases} r_j \text{ if final state} \\ r_j + \gamma Q'(s_{j+1}, \mu'(s_{j+1}|\theta^{\mu'})|\theta^{Q'}) \text{ otherwise} \end{cases}$
        Update critic minimizing the loss $L = \frac{1}{m}\sum_j (y_j - Q(s_j, a_j, \theta^Q))^2$
        Update the actor policy with sampled policy gradient

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

        Update target networks

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

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

    end for
end for

Chapter 5

Robot Platform

5.1 Chapter overview

The aim of this chapter is to provide the reader with a basic knowledge about the robotic development framework. The chapter is opened with a brief introduction to robotic platforms. An overview of laser distance sensors and cameras is shortly depicted, focusing on what has been actually used in this project. The robot models and the software tools used for the simulations are described in the second part of the chapter.

5.2 Introduction to robotic platform

A *robot* is a programmable machine able to accomplish a specific set of tasks. More in detail, the term *robotic platform* is often used to indicate the ensemble of the **hardware** elements composing the robot and the **software**. The hardware mainly involves the mechanical the links and the joints, the motors, the electronic boards and the sensors. The software is usually composed of a firmware to control the robot and additional tools necessary to manage properly the computational resources of the hardware part. The architecture of a robot may vary in a wide range of possibilities, according to the desired application.
Among the most popular configurations there are the followings:

- Industrial robots: inspired to a human arm, realized with a set of links and joints.
- Mobile robots: also called Unmanned Ground Vehicles (UGVs), usually wheeled robots adopted for service robotics, logistics and exploration purposes.

- UAVs: Unmanned Aerial Vehicles, often called drones. Quadcopters are the most popular ones.
- Humanoid robots: full body robots inspired by humans.
- Others: soft robots, biomimetic robots inspired by animals, exploration robots, etc.

In this thesis, we take in consideration wheeled mobile robots for service robotics. Robots may have different levels of autonomy. An AI framework such as the DRL agent developed in this project aims to achieve a full autonomy in the navigation task. This aspect is mainly related to the software part of the robotic platform. The hardware can be a custom realization of the developer, as well as a standardized commercial platform. In the next sections the robotic platform used for this thesis are shortly described.

5.3 Sensors

Sensors are the hardware devices responsible of the collection of data from the environment and the robot itself. The choice of the most suitable sensor is a relevant design factor, both from an hardware and a software point of view. In fact, the information contained in sensor data affects the efficiency of an algorithm according to the data dimensionality and accuracy. For what concerns an autonomous navigation system, sensors are mainly responsible of the measurement of the robot pose, the goal position and the obstacles distance. For additional tasks, sensors for people or object detection can be also integrated. Here two families of sensors are taken in consideration:

- *Laser Distance Sensors* (LDS): 2D LiDAR for obstacle detection.
- *Visual sensors*: Depth Camera for obstacle detection.

These perception technologies are briefly described in this chapter, whilst their implementation will be detailed in 6.

5.3.1 Laser distance sensors (LDS)

Laser Distance Sensor (LDS) is a family of sensors commonly used in robotics. Among them, Light Detection and Ranging (LiDAR), Laser Scanner and Laser Range Finder (LRF) are popular examples. LDS exploits laser technology to measure the distance from the surrounding objects, which reflect the laser beam. LDS are particularly useful whenever a real-time information of the environment is needed by the robot to accomplish its task. For example, in obstacle avoidance it is fundamental to receive such information in time. Laser based sensors are typically

composed of a rotating support, the laser generator and a mirror. By focusing on 2D **LiDAR**, which is the one used in this thesis, the support rotates in a 360° angular range, obtaining the distance measures of points on the same horizontal plane.
A competitive advantage offered by LiDAR data is that they are computationally affordable. The measured points are often represented by a vector of relatively small dimensions, differently from images. However, a LiDAR based perception system can represent a limitation with respect to visual based solutions. Indeed, the data are collected only on a 2D plane and sometimes a richer spatial information is needed. This is a particularly relevant issue for navigation systems, since objects with complex shapes can represent a challenging problem. Another critical aspect is related to the reflection properties of the obstacle surface. Transparent materials can dramatically reduce the efficiency of such sensors. In general, the precision of the measurement is correlated to the distance from the obstacles.

5.3.2 Visual sensors: Cameras

Robotic vision is probably the most popular perception system used to let the robot interact with the surrounding environment. It is inspired by the human vision, which can be considered an incredibly sophisticated perception system. In fact, the human visual system is able to interpret the information embedded in the light received by the retina. This is an extremely complex task that all computer vision systems aims to accomplish. In an analogue way, images are exploited in robotics to extract information about the environment. In general, images provide a detailed level of knowledge of the scene with respect to 2D laser points. Visual data can be both 2D, if common 3-channel RGB colour images are used, and 3D. In this last case, the depth information is added with an additional channel. Practically, for the majority of robotic tasks depth information is the most valuable one, especially when dealing with the motion control of the robot. RGB images results to be precious for object detection and segmentation tasks.

5.3.3 Depth Camera

Depth cameras are particular visual sensors that provide information about the distance of points in the captured scene. In fact, depth images are single channel, grey-scale images. Each pixel of the image tensor is a numerical value associated to the distance of that specific point with respect to the camera frame. As already discussed in chapter 2, depth cameras are increasing their diffusion in robotic applications thanks to their convenient trade-off between computational cost and information.
There are different types of technology which are able to provide a depth map.

Time of Flight (ToF) cameras exploits light reflection similarly to a radar or a LiDAR. A light pulse is generated to illuminate the scene and a sensor is used to detect the reflected radiation. Differently from point-to-point LiDAR data, the light pulse is unique and provide a single depth image without the need of a moving mechanism. Although the method of this relatively recent solution is effective, the necessary hardware is quite expensive.
Structured light cameras are an alternative technology. This time, the object is illuminated by a designed structured light and a single depth image is obtained. A simple image sensor is able to build the depth map by receiving the reflected light pattern from the object. The hardware required for this technology is generally cheaper than ToF cameras.
Stereo-vision is the technology more inspired by the human visual system. In fact, two image sensors are used to generate the depth map. The triangulation method is used to calculate the distance between the two lens of the camera and the object. Infrared rays are usually emitted by a projector and the two image sensors are responsible of the depth map reconstruction. This is also the working principal of Kinect depth cameras such as the Intel RealSense, already mentioned in chapter 2.

5.4 TurtleBot3

In the previous section, an overview of robotic platforms has been given. At this point, it is possible to introduce one of the most popular platform among robotic developers: TurtleBot3.
TurtleBot can be considered a standard for robotic platforms. It is based on open source Robot Operating System (ROS) and it has been developed with a great focus on research and educational purposes. In fact, on the one hand it offers the tools to develop professional projects, on the other hand it is an easy instrument to let beginners make the first steps in robotics. The actual available version of the platform is the TurtleBot3, developed in 2017 from the collaboration of the firms Open Robotics and ROBOTIS. TurtleBot3 platform aims to offer a low-cost valuable kit which also allows for a further customization desired by the developer. There are three available mobile robots in TurtleBot3: Burger, Waffle and Waffle Pi. All of them mount a Single Board Computer (SBC), an embedded controller OpenCR and Dynamixel actuators. The TurtleBot Burger is characterized by a smaller mechanical chassis. From the perception point of view, it is only equipped with a 2D LiDAR having a 360° angular range and an angular resolution of 1°. In addition to the LiDAR, the TurtleBot Waffle also makes use of an Intel RealSense for 3D perception. A simpler solution is the Raspberry Pi camera used by the Waffle Pi. In this thesis project, only the virtual models of the TurtleBot Burger and Waffle are tested in simulation. They are shown in Figure 5.1.

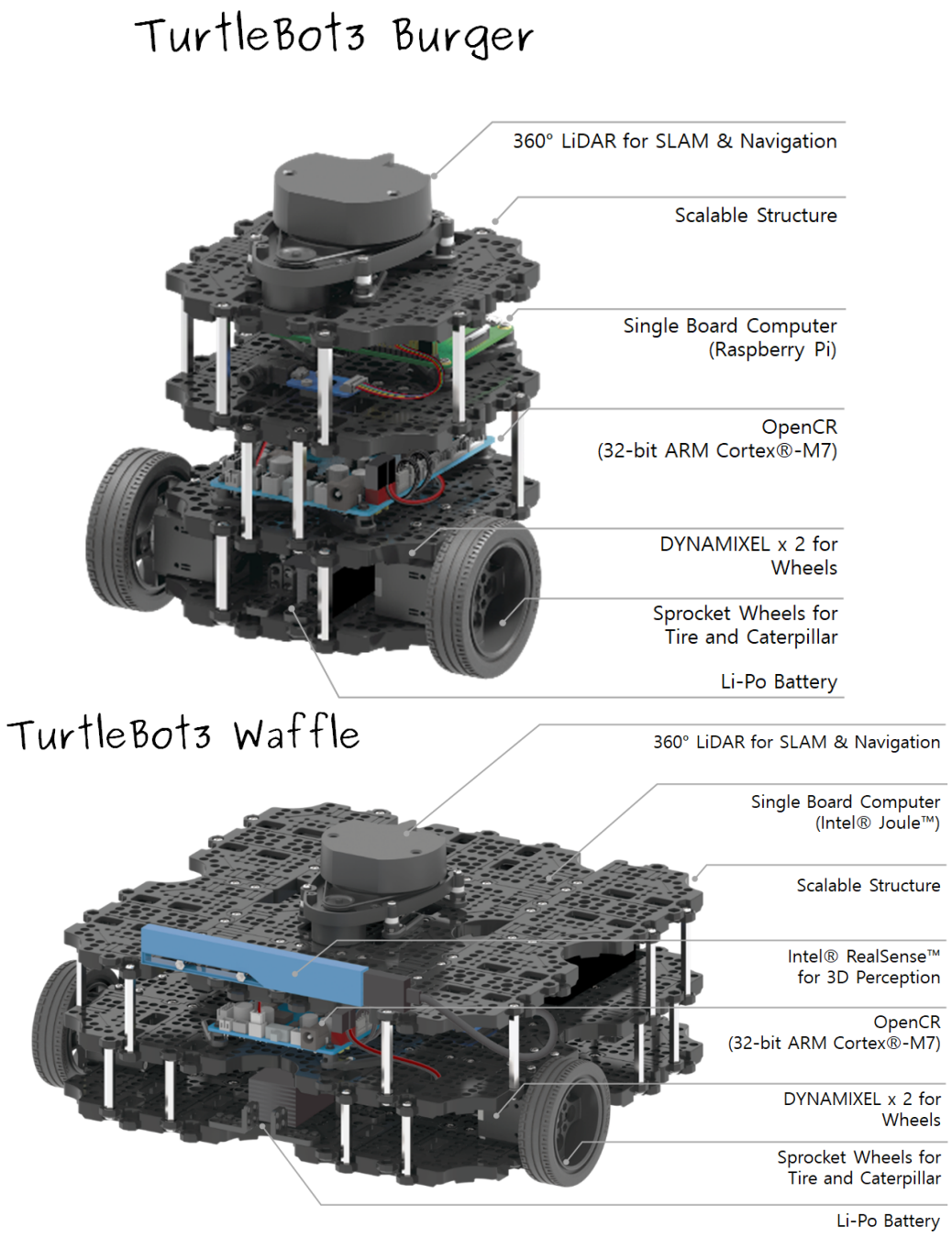

Figure 5.1: Hardware component description of TurtleBot Burger (top) and TurtleBot Waffle (bottom).[26]

5.4.1 Actuators

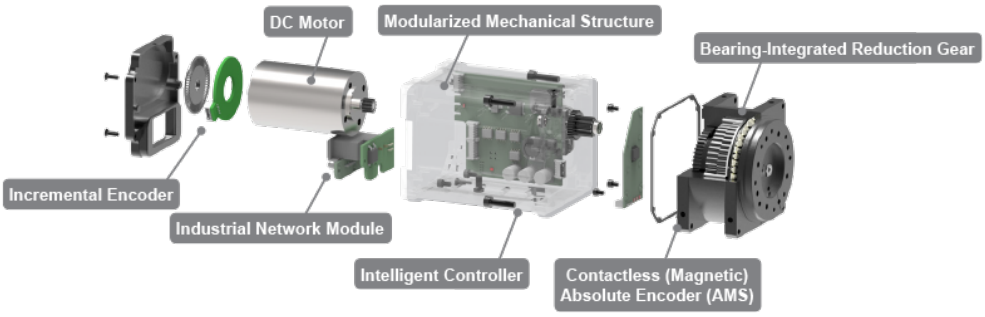

Figure 5.2: Dynamixel actuator components.
[27]

TurtleBot3 platforms uses electric motors to actuate the motion at the wheels. In particular, Dynamixel is the actuator developed by ROBOTIS for robotic developers. The components of the Dynamixel actuator are shown in Figure 5.2. Beside the classic mechanical parts proposed with a modular architecture, it comprises an incremental encoder sensor and a controller board. The combination of the DC electric motor with the controller enables a direct easy implementation and a wide variety of applications. A PID control strategy is used for feedback loops, in order to guarantee the desired frequency, position, velocity and current. Dynamixel aims to minimizing the current consumption for an optimal battery usage. The target quantities for the actuators are provided by the software running on the robotic platform. Different Dynamixel actuators can be chosen according to the performance required.

5.4.2 OpenCR

OpenCR is the embedded system which takes care of controlling the operations of the TurtleBot3 robot. It has been specifically designed to offer a great hardware flexibility and to be compatible with ROS. First of all, it is convenient to explain that an embedded system is a special purpose system composed of: an hardware part, a real-time operating system (the basic software part) which is responsible of the management of the hardware resources, and a software application running on top of it. OpenCR is based on an ARM Cortex-M7 microcontroller, belonging to the STM32F7 chips. It is a powerful microcontroller, that is also able to compute data with floating point unit. The board offers the possibility to connect many peripherals, for example to enable the communication with sensors such as LiDAR and cameras or with the wheels actuators of the robot. The chip also contains 3-axis accelerometers, magnetometers and gyroscopes.

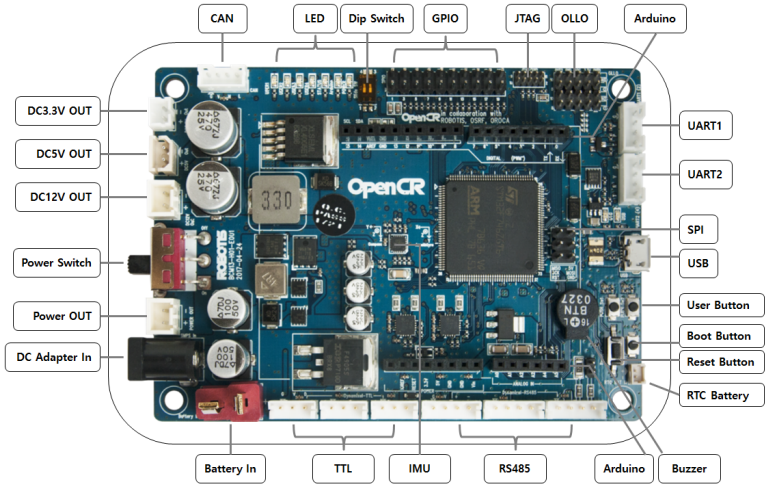

Figure 5.3: OpenCR embedded controller.
[26]

5.4.3 Intel RealSense R200

The Intel RealSense R200 camera is the standard camera model available for the TurtleBot3 Waffle, also in simulated environment. The real hardware camera belong to the stereo-vision camera technology. In fact, it has two IR images sensors and a IR projector. In addition to that, it has a RGB camera. Hence, it is able to provide both RGB colour images and single channel depth images at different resolution and frequency. In Figure 5.4 the RealSense R200 is shown together with its technical specifications.
For what specifically concerns its usage in this project, it is important to point out that the resolution of 640 x 480 indicated for depth images is the maximum available for the physical camera. In simulation, a resolution of 320 x 240 is set as default.

5.5 Software tools

In this section the software tools used to develop the project are described. First of all, this work has been carried out on a machine with Ubuntu, an open-source Linux based operating system. The version used is the Ubuntu Bionic Beaver (18.04). Beside this, the core of the software that will enable the deployment of the application on a real robot is composed by ROS (Robot Operating System) packages. For this work, the second official version of ROS is used, ROS2. It presents several improvements at the building level with respect to the first one. The distribution used is the ROS2 Dashing Diademata. The code for the application, namely for

Items	Specifications
RGB Video Resolution	1920 x 1280, 2M
IR Depth Resolution	640 x 480, VGA
Laser Projector	Class 1 IR Laser Projector (IEC 60825-1:2007 Edition 2)
Frame Rate	30 fps (RGB), 60 fps (IR depth)
FOV (Field-of-View)	77° (RGB), 70° (IR depth), Diagonal Field of View
Range	0.3m ~ 4.0m
Operating Supply Voltage	5V (via USB port)
USB Port	USB 3.0
Dimensions	101.56mm length x 9.55mm height x 3.8mm width
Mass	Under 35g

Figure 5.4: Intel RealSense R200 labels and technical specification.[28]

the DRL agent and the environment, has been developed using Python. It is a high-level programming language which is largely spreading out thanks to its great flexibility and easiness of usage. In fact, Python supports multiple programming paradigm such as structured, object-oriented and functional programming.

5.5.1 ROS

ROS is a free and open-source meta-operating system for robotics. More in detail, it is a middleware software platform, i.e. a collection of tools which lets interact effectively the hardware and the software application. As such, it provides hardware abstraction and packages management. On the other hand, it is true that the ROS monolithic microkernel is very limited. ROS can be therefore defined as a *thin tools-based* operating system. Each ROS component is executed separately from the others allowing for a better robustness. Scripts can be launched as standalone executable. This is a design choice motivated by the intention to leave the developers an high freedom and the possibility of customizing the system.

Moreover, ROS is not a real-time operating system, even though a low latency is a key requirement in robotic applications. ROS also supports multiple programming languages.

Beside these aspects, it is interesting to briefly describe how does ROS work. ROS processes can be represented as a *graph* composed of connected **nodes**. For this reason, it is said to have a distributed **peer-to-peer** (P2P) architecture. Each node of the graph is a process in execution. Figure 5.5 shows an example of ROS graph.

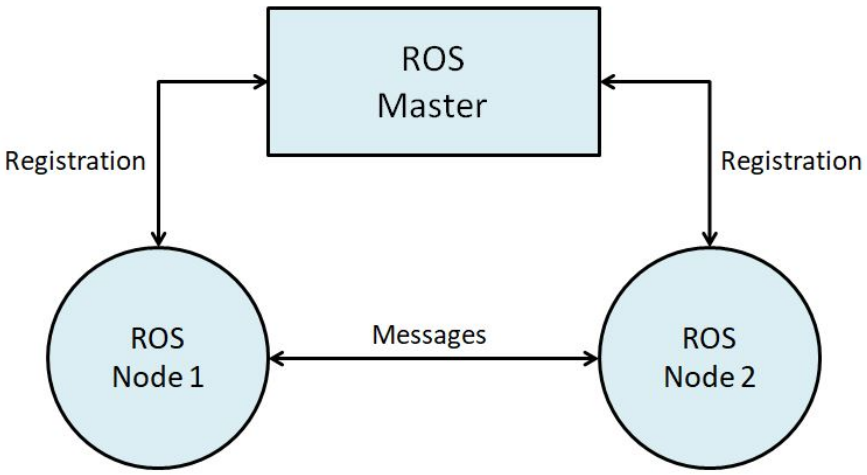

Figure 5.5: ROS execution graph with nodes and Master.

A particular process called *ROS Master* is responsible of registering the nodes to itself and to enable the communication between nodes. These can send and receive messages in different ways, among them the principal ones are:

- **topics**: they are buses for messages characterized by a specific namespace. The communication through topics is based on a mechanism of publisher/-subscriber. An anonymous *publication* of messages on a specific topic allow a node subscribed to that topic to access to the messages.

- **services**: they are another way to communicate actions between nodes. In this case, a node will advertise the service (server) and the other one will receive the service (client).

The ROS building structure is schematized in Figure 5.6.

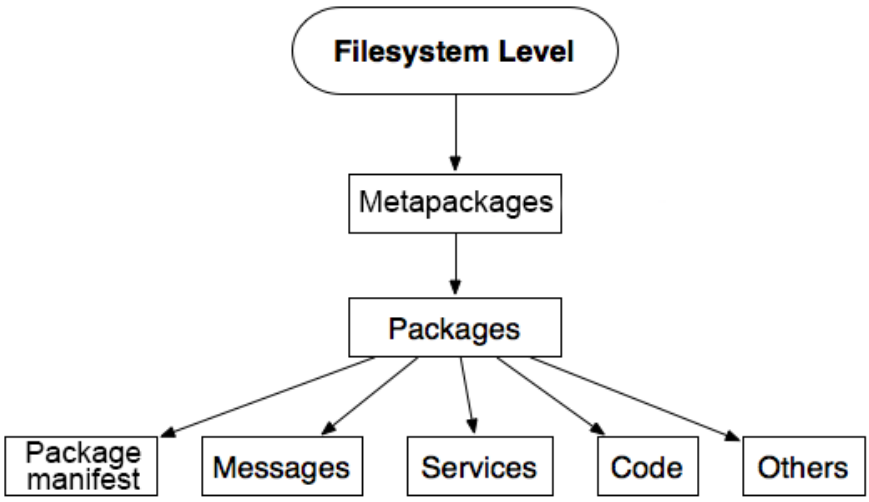

Figure 5.6: ROS building organization.

Where **packages** are the atomic units of ROS. They contain the nodes as well as all the configuration files and libraries associated. The package manifest contains licenses, dependencies and other information about the package. Packages are stack in metapackages and grouped according to their specific purpose.

5.5.2 Gazebo

Gazebo is the software used to simulate the virtual environment. The great success of Gazebo is mainly related to its open-source nature and to the necessity to have a 3D virtual environment to simulate robots behaviour before test them in the real-world. It is very common to use it in combination with ROS. Gazebo exploits high performance physics engines and solvers like ODE, Bullet, etc, to offer high-quality rendering. Lighting and textures are also represented realistically. More importantly, it comes with a wide variety of models of robotic platforms such as TurleBot, Pioneer2 DX and many others. Virtual copies of sensors are also available, from laser scanner to cameras. Contacts and collisions between models are also simulated. Hence, it is easy to set up a realistic indoor scenario such as an office or an apartment and let the robot navigate and interact with the entire environment. Gazebo 9 is the specific version used for this project.

5.5.3 Machine Learning tools

The development of a navigation system with a DRL agent requires a software application where the artificial neural networks can be built as computational models. For such a purpose, two software frameworks are imported into a Python script, which is executed inside the ROS graph as a node. They are TensorFlow and Keras.

TensorFlow is an open-source software platform which has obtained a great popularity among Machine Learning developers. It was firstly created by Google Brain team and released in 2015 for Google project concerning deep learning. TensorFlow 2 has been recently released (September 2019) and it is used in this work. It offers a complete set of tools useful to create a full computing pipeline for neural networks. TensorFlow APIs (Application Programming Interfaces) are exploited to: build the neural networks models, process data, compute gradients and train models during the simulation. More in detail, Gradient Tape is the API used for automatic differentiation and it allows to compute gradients and set up a customized training. Furthermore, for the training of the DRL agent, tensorflow-gpu is installed on a workstation to manage the GPU computational resources.

Keras is a high-level API for deep learning written in Python. Keras runs on top of TensorFlow, or in other words it uses TensorFlow as back-end. It allows to easily instantiate a deep learning model defining each layer, activation function, etc.
Popular APIs offered by Keras are the Sequential model and the Functional API model. The last one allows to easily build models with multiple inputs.

Chapter 6

The navigation system

6.1 Chapter overview

In this chapter, details about the autonomous navigation system and the implementation in a virtual environment are provided. Everything has been carried out in simulation only, although both robot components and sensors are virtual models of hardware devices physically available at the PIC4SeR centre. The core of the chapter is devoted to the presentation of the proposed solution, focusing on the sensor data and on the actor-critic model chosen for the DRL agent. First, the navigation framework with LiDAR data is presented. Finally, the visual based navigation with depth images is described.

6.2 Simulation setup

For the development of the project, a workstation has been bought and used to locally train the neural networks in simulation. The computational resources of the workstation consist in 32 GB of RAM, a NVIDIA RTX 2080 Super (8 GB) GPU and a CPU Intel core i7-10700. As in the majority of deep learning applications, also in this case the Graphic Card plays a central role and the whole training process has been designed and tuned according to its resources. Time of execution and memory are the main constraints that have to be sharply respected.

All the simulations have been run in Gazebo virtual scenarios. Different stages are exploited to let the DRL agent learn how to navigate efficiently in environments with the presence of different obstacles. The robot model is spawned inside the virtual world, where its behaviour can be visually monitored. Also sensor data are shown and collected in the dedicated ROS topics. Figure 6.1 shows an example

of basic virtual environment in Gazebo with the TurtleBot Waffle model moving around. Both the LiDAR rays (blue lines) and the RealSense camera view are visualized in real-time.

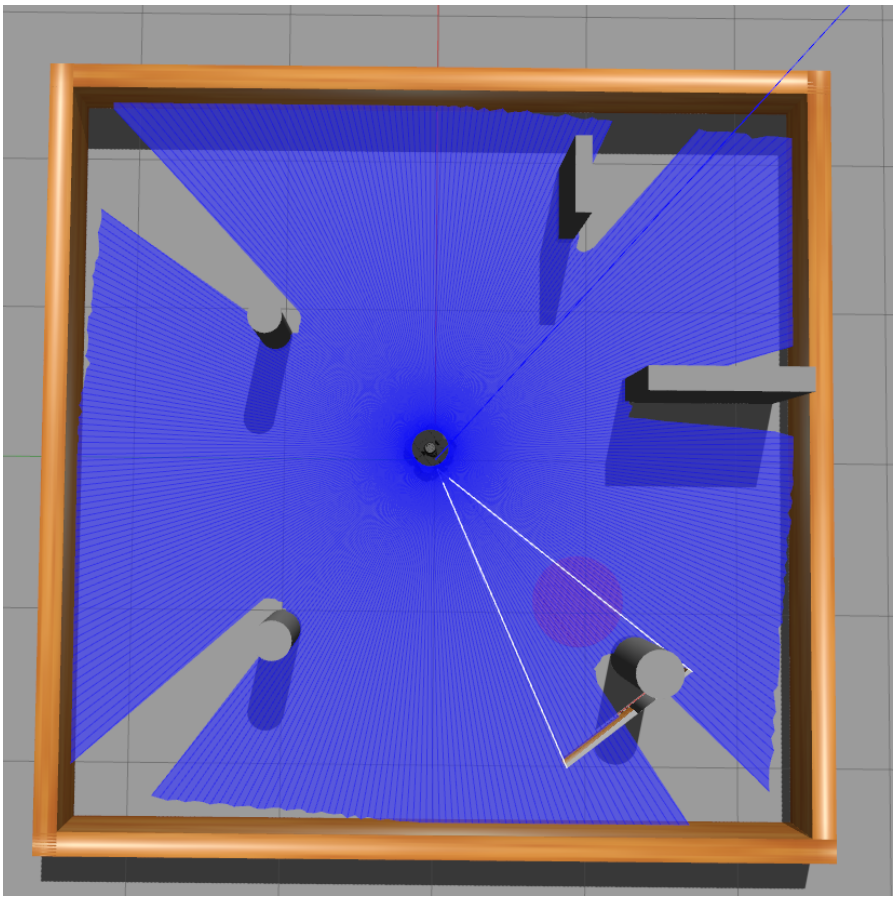

Figure 6.1: An example of Gazebo simulation with waffle robot.

Simulations are structured in episodes, since a Reinforcement Learning framework need to be reproduced to train the agent. According to the complexity of the neural network used for the agent, the reward function and the difficulty of the stage, the simulations may require a huge amount of episodes and time to be successful. For this reason, it is often convenient to speed them up, checking if the time needed to execute the code allows to do it. This is done exploiting the physics functionalities of Gazebo. A feasible configuration results to be obtained increasing the real-time update of the simulation to 1.200 (simulations run at a speed of 1.2x).

At this point, it is convenient to briefly illustrate the organization of the ROS2 package developed at the PIC4SeR centre to handle reinforcement learning frameworks in ROS. It is called 'pic4rl' and it aims to provide the software application needed to train a RL agent, with a generic robot model in a Gazebo simulation. The package structure exploited in this work is illustrated in figure 6.2. In parallel to the simulation running on Gazebo, two connected nodes exchange messages: 'pic4rl training' and 'pic4rl gazebo'.

- 'pic4rl training': it contains the TensorFlow functions to instantiate and train the neural networks with the DDPG algorithm, as well as the code for the Environment response.
- 'pic4rl gazebo' it mainly handles the service to reset the world and to receive sensor data from the simulated scenario.

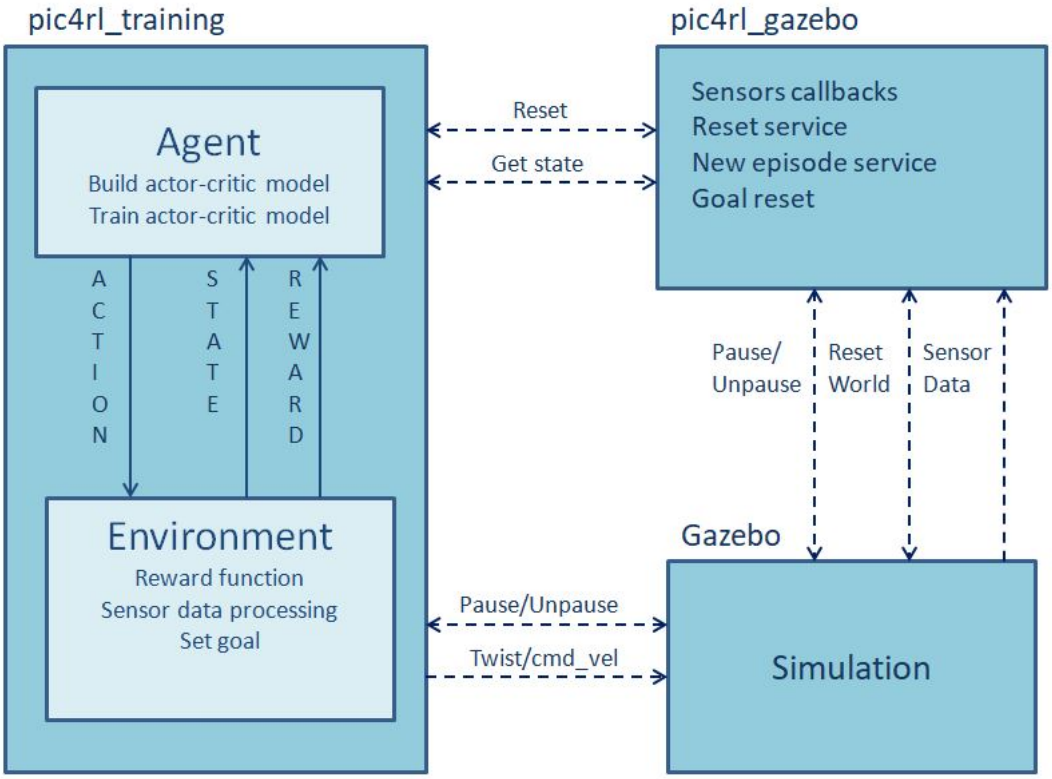

Figure 6.2: Pic4rl package: nodes organization.

The basic cycle executed by the code at each temporal transition can be depicted as shown in figure 6.3. Once an episode of the simulation has been set, defining the goal position, the actor model selects an action a_t based on the information

contained in the received state s_t. The selected action, which is always expressed as a tuple of linear velocity and angular velocity, is executed by the robot in the virtual world. The environment samples a reward according to the designed reward function and checks if the episode has to be terminated or not. There are three possible events for the end of an episode:

- *Goal*: if the robot reaches a sufficiently small distance from the goal coordinates. This tolerance is indicated with δ_g and the goal is considered reached if $\delta_g < 0.2m$.
- *Collision*: if the robot collides with an obstacle in the virtual world. A minimum distance of $\delta_c = 0.22m$ is accepted, otherwise the collision event is detected.
- *Timeout*: after 500 temporal steps the episode is terminated in any case.

If none of those conditions is verified, the environment receive and process sensor data from Gazebo. Then, the information concerning the position and orientation of the robot with respect to the goal are updated and the new state s_{t+1} is sent back to the agent.

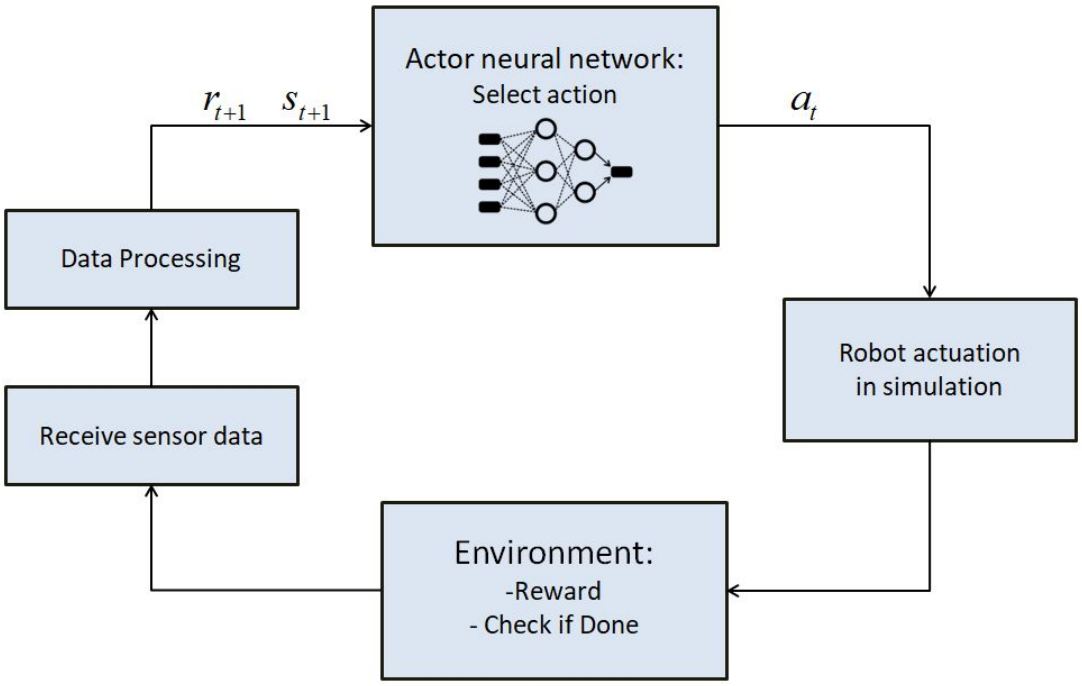

Figure 6.3: Code logic basic scheme at a generic temporal transition.

6.3 LiDAR-based navigation

The first type of solution implemented in simulation is the LiDAR-based autonomous navigation. Although it is not the main goal of this project, it is considered a fundamental step in order to acquire experience in handling reinforcement learning systems coupled with ROS. Moreover, it constitutes a precious source of comparison for what specifically concerns the efficiency of navigation with different sensor data. The analysis of the proposed implementation begins from the description of the data structure exploited. Then, both the neural networks and the reward functions are illustrated with respect to the training process.

6.3.1 Data filtering

Odometry is the term used in robotics to indicate the usage of motion sensor data to compute the position of the robot in time. This is done thanks to the ROS standard TurtleBot3 packages. The change of the position of the robot is computed with respect with its initial reference frame. These data are advertised by the ROS topic */odom*.

For the purpose of the navigation system, odometry is necessary to extract information about the distance and the orientation of the robot with respect to its target. Only the information related to the orientation (yaw) of the robot are processed. In particular, they are needed to compute at each time step the *heading* angle, i.e. the angle between the axis of orientation of the robot and the one passing from the target point. Hence, the **distance** from the goal can be easily computed with:

$$d = \sqrt{(x_g - x_r)^2 + (x_g - x_r)^2}$$

According to the scheme in figure 6.4, the **heading** angle can be computed in two steps. First, the angle between the vertical axe of the robot and the goal is obtained:

$$\alpha = atan2(y_g - y_r, x_g - x_r)$$

Then, the heading γ is computed exploiting the known yaw angle of the robot, according to the angular range:

$$\gamma = \begin{cases} \alpha - yaw \;\; if \;\; -\pi \leq \alpha - yaw \leq \pi \\ \alpha - yaw - 2\pi \;\; if \;\; \alpha - yaw > \pi \\ \alpha - yaw + 2\pi \;\; if \;\; \alpha - yaw < -\pi \end{cases}$$

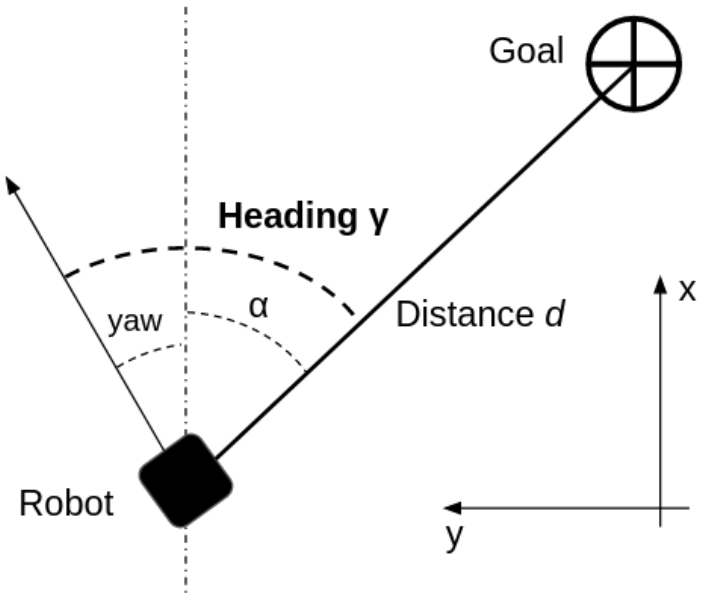

Figure 6.4: Scheme of heading angle between the robot and the goal. Angle α and yaw are exploited to compute it. Goal distance is also indicated.

LiDAR data are the sensor data used in this implementation to enable the obstacle avoidance ability of the robot. The raw LiDAR measurements are composed of 359 distance values stack in an array. The first value indicates the measure of the distance in front of the robot. Then, other measures proceed all around the robot in a counter-clockwise order. An array of 359 values is considered too computationally expensive for such an implementation. Hence,the measurements are filtered before to use them as input for the neural network. Different dimensionality of the array of LiDAR points are tested, always keeping a FOV of 360°: 10, 24, 36 and 60 all-around points. Increasing the dimensionality of the array, the convergence of the algorithms is sometimes compromised. In fact, it becomes more difficult for the neural network to distinguish between LiDAR measurements and goal distance. An array of 36 LiDAR filtered measurements results to be a good trade-off between the navigation accuracy and the training effort.

When the 359 measurements are received in the environment node from Gazebo, the minimum distance value is selected for each angular interval of 10°. This is a simple effective data filtering, which aims to guarantee an effective detection of obstacle all around the robot. With an array of equally-spaced points, a narrow obstacle may be not perceived at all.

To sum up, the system receive a processed data structure composed of:

- Goal distance value.
- Goal angle value.
- Filtered LiDAR points.

This ensemble of information are stack together to compose a state s_t at a given temporal instant t. The resulting array also represents the input of the actor neural network, described in the following paragraph.

6.3.2 Actor-Critic neural networks

The navigation system is trained using the DDPG algorithm, to approach with a DRL agent the continuous control task. The actor-critic architecture characteristic of DDPG has been explained in chapter4. Here, it is convenient to remark that the **actor** neural network is the one selecting actions for the robot, hence it is the one responsible for the control of its motion. The actor network is therefore composed of an input layer which receives LiDAR points, goal distance and heading. Then, three fully-connected layers with ReLU activation function are set. The output is passed to two different single-neuron layers used to map the linear and the angular velocities for the robot. A *sigmoid* activation function, scaled in the interval [0,0.2] is used for the linear velocity. Hence, the robot has no possibility to move backward and its maximum linear speed will be equal to 0.2 m/s to ensure a safe future hardware implementation. The angular velocity is instead obtained from an hyperbolic tangent function. Therefore, its numerical value will be naturally comprised in the interval [-1,1]. The two velocity signals are finally concatenated to compose the predicted action a_t. The robot controller will takes care of actuating the velocity command published on the /cmd_vel ROS topic, with the following specifications:

- A linear velocity $v \in [0,0.2] m/s$
- An angular velocity $\omega \in [-1,1] rad/s$

The **critic** network presents a general analogue structure to the actor. The main difference is an additional input branch to receive the actions to be evaluated. After some fully-connected layers, the output neuron with a Linear unit is used to predict the Q-value.
Actor and Critic models for LiDAR-based navigation are shown in Figure 6.5 for a better visualization.

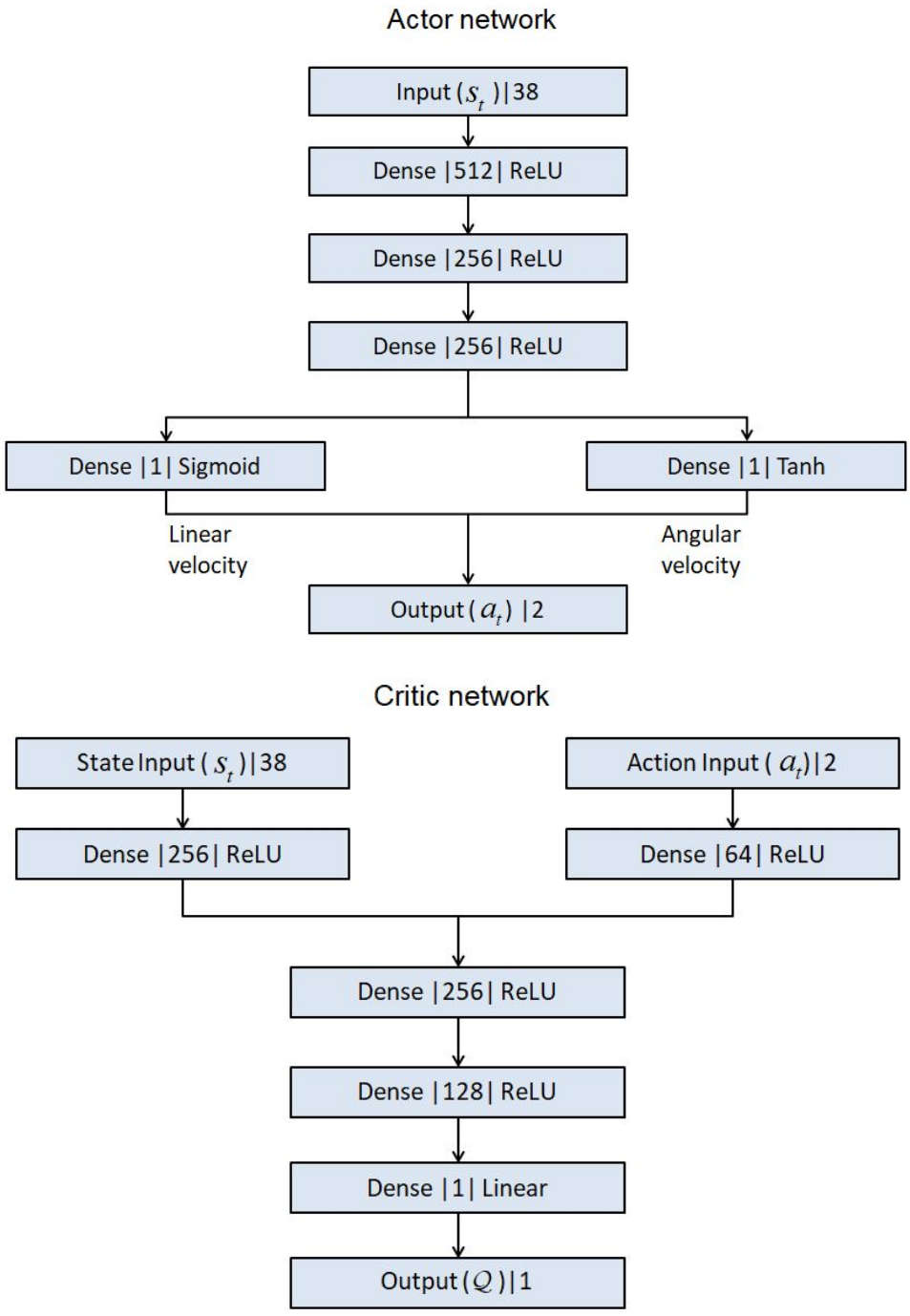

Figure 6.5: Actor and Critic neural networks for the LiDAR-based navigation system.

6.3.3 Reward function

The design of a suitable and effective reward function is one of the tough challenges of this work. There are no clear and standardized indication in literature concerning the design of a generic reward function, but it strongly depends on the specific application of the RL agent. A general structure for the reward function is composed of a fixed contribution, assigned at the end of each episode, and a sparse reward. The last one is a smaller evaluation of the agent behaviour at each temporal step. However, it has a great relevance in the learning process.

For this work, different reward function have been tried so far. Here a selection of the relevant candidates is proposed. From an high-level perspective, the reward function is shaped as follow:

$$r = \begin{cases} r_g & \textit{if Goal} \\ r_c & \textit{if Collision} \\ r_b & \textit{otherwise} \end{cases}$$

For LiDAR-based navigation, the fixed rewards related to final events such as Goal or Collision are kept constant. In particular the values $r_g = 100$ and $r_c = -10$ results to be as simple as effective for the convergence of the DDPG algorithm.

A greater focus must be devoted to the choice of a suitable sparse behavioural reward r_b. The followed approach consists in combining two terms: one related to the heading of the robot and one focused on its motion towards the goal. The main idea behind that choice is that the robot should learn to navigate keeping the goal aligned with its direction of motion to minimize the total path.

The heading reward is shaped using the following formulation:

$$r_h = \left(1 - 2\sqrt{\left|\frac{\gamma}{\pi}\right|}\right)$$

where γ is the heading angle between the goal and the direction of motion of the robot, obtained from odometry. Differently, for the distance based reward different shapes can be chosen. Here some examples are provided:

$$r_{d1} = 2\left(2\frac{d_{t=0}}{d_{t=0} + d_t} - 1\right)$$

where $d_{t=0}$ is the initial distance from the goal, at the beginning of the episode, whilst d_t is the actual one.

$$r_{d2} = d_{t-1} - d_t$$

$$r_{d3} = 2 - 2^{\frac{d_t}{d_{t=0}}}$$

A first try is done with the following sparse reward:

$$r_b = r_h + r_{d1}$$

However, it results to be successful only with a state of very small dimensionality. For example with an array of up to 4 LiDAR measurements. Hence, a different combination is tried out and a further tuning with empirical numerical coefficients is performed. The resulting final reward function used to train the agent is:

$$r = \begin{cases} r_g = 100 & \textit{if Goal} \\ r_c = -10 & \textit{if Collision} \\ r_b = 0.8(r_h) + 30(r_{d2}) & \textit{otherwise} \end{cases}$$

6.3.4 Training process

As already mentioned at the beginning of the chapter, the training process evolves alongside the simulation in virtual environment. A sufficient number of episodes is carried out until the algorithm converges. More in detail, for the LiDAR-based navigation different stages are used to train the neural networks. As a first attempt, the agent is trained in standard scenarios with obstacles of different shapes. The first one is an empty space with only outer walls, the second presents four columns and a relatively wide space of maneuver, whilst the third one is mainly composed of walls with narrow passages (see Figure 6.6). Then, the training process is also experienced on a custom virtual world with obstacles realized from scratch. Similar results are obtained.

ADAM optimizer is used to update parameters of both actor and critic networks. The complete set of hyperparameters for training and the simulation settings are listed in Table 6.1.

Learning rates for actor and critic networks are set to different values. This choice results to be popular in literature and it is considered reasonable since the critic works effectively when it starts to correct the action behaviour as soon as possible in the learning process. A slightly lower level of accuracy can be accepted for the critic since it will be the actor to control the robot motion. Train starts after 64 episodes in which the robot accumulates experiences through exploration. This is necessary because mini-batches for training are sampled randomly from a replay memory buffer that must be filled with tuples of transitions $(s_t, a_t, s_{t+1}, r_{t+1}, done)$.

Parameter	Value
Training hyperparameters	
batch size	64
train start	64
target networks update start	128
replay memory buffer maxlen	10^6
actor learning rate	0.0001
critic learning rate	0.0008
tau	0.9
starting epsilon	1
minimum epsilon	0.05
epsilon decay	0.998
discount factor	0.99
Navigation settings	
lidar points	36
max linear speed	$0.2m/s$
max angular speed	$1rad/s$
Simulation settings	
max number of episodes	5000
time step	0.001s
max update rate	$1.200s^{-1}$
timeout	500 steps

Table 6.1: Hyperparameters and simulation settings for LiDAR-based navigation.

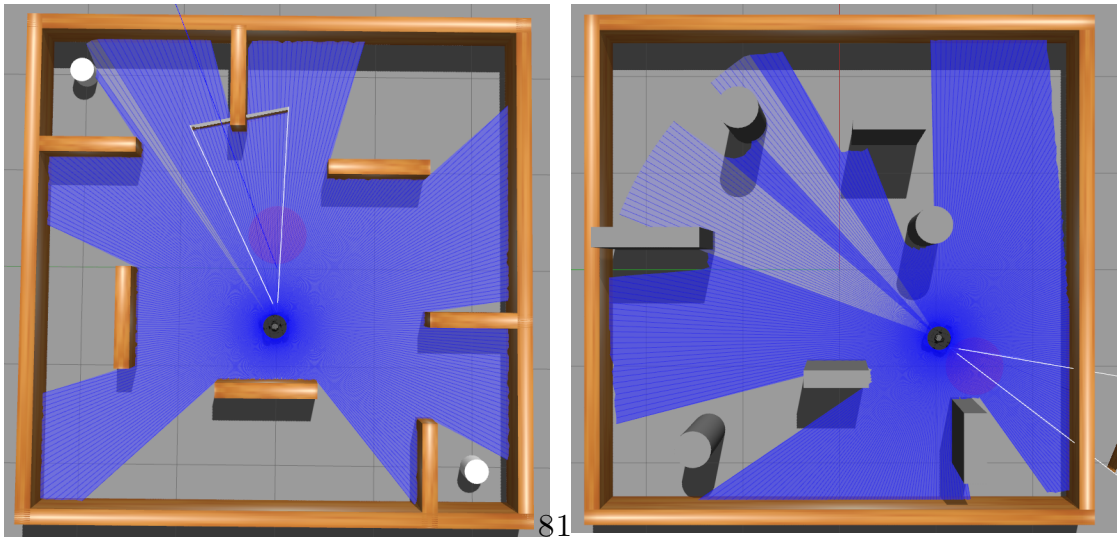

Figure 6.6: On the left the final standard scenario with static columns and walls. On the right a custom training scenario realized from scratch.

As explained in chapter 4, devoted to Reinforcement Learning, exploration must be guaranteed in order to find an optimal policy. This is done by using a stochastic $\epsilon - greedypolicy$ for the selection of random actions according to the value of the parameter ϵ. It is decreased with a decay policy such that after a suitable number of episodes the randomness in the action selection is reduced and the robot acquires more control on its motion. As already mentioned in the DDPG theoretical description, τ is a constant used for the soft update of parameters in the target networks, for an improved robustness of the algorithm. The *discount factor* is used for the critic network training to compute the target values according to the DDPG algorithm.

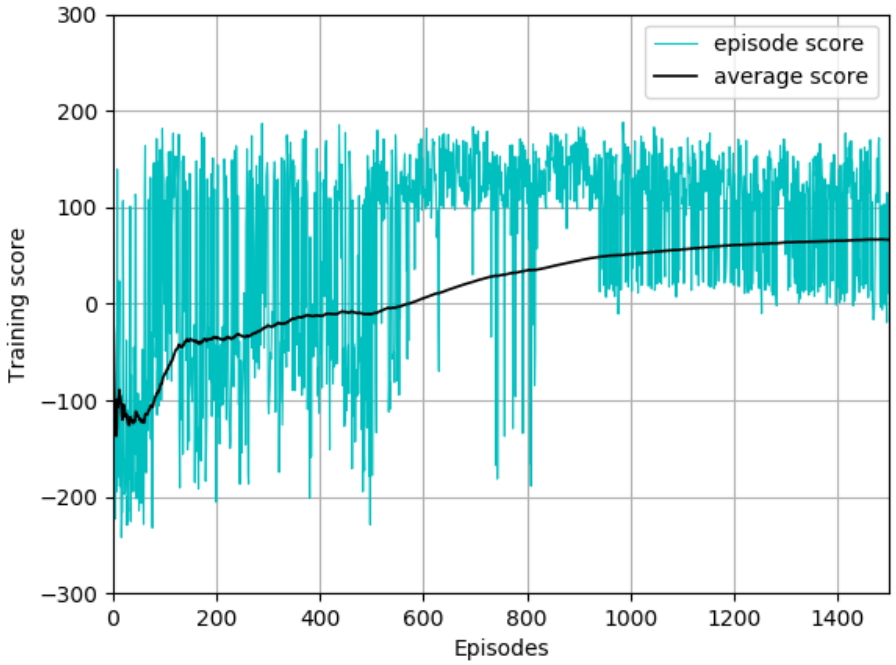

Figure 6.7: Learning curve with LiDAR sensor in a successful simulation with the described reward function. After episode 800 the convergence of the algorithm is stable and the goal is always met. From episode 950 static obstacles are added to the scene.

6.4 Visual based navigation

The idea of a visual based navigation is principally considered to tackle the difficulty in perceiving obstacles of complex shapes, which is a limitation for the LiDAR sensor. Cameras provide a greater amount of details related to the scene, although in a limited FOV. In particular, depth images are exploited in this implementation to provide the robot with information about obstacles. As already said in the previous chapters of the thesis, depth images are single channel grey-scale images, containing distance values in the pixels. The choice of a visual based navigation system requires for some modifications in the training process with respect to the previous implementation.

6.4.1 Data processing

The main concern in this new navigation system is related to the different data format. With respect to the previous implementation, **Odometry** data are obtained and processed in the same way. Heading angle and distance from the goal are computed using the same equations described for the LiDAR navigation.

Depth images are received by the 'pic4rl training' node thanks to the subscription to the ROS topic dedicated to the Intel RealSense R200 camera. In order to access to raw depth images, it is only required to add a plugin in the Gazebo model of the waffle. The RealSense is able to provide three different data formats:

- RGB images: colour images, published on the ROS topic /intel_realsense_r200_rgb/image_raw.
- Depth images: grey-scale images, published on the ROS topic /intel_relasense_r200_depth/image_raw.
- Depth point cloud: spatial visual representation of obstacle surface with points realized according to the depth map, published on the ROS topic /intel_relasense_r200_depth/points.

Only depth images are used in this implementation. Messages containing the images are imported in the environment and processed thanks to a callback function. The CVbridge library, from the OpenCV collection of tools for Computer Vision, allows to decode the image from ROS messages to a numpy array. Raw depth images provided by the RealSense topic have a resolution of 240x320 (height x width).

At this point, it is convenient to recall some key aspects of the the DRL framework in order to understand the difficulty behind this task. First of all, during training mini-batches are sampled from the replay memory buffer. Transitions $(s_t, a_t, s_{t+1}, r_{t+1}, done)$ are stored at each temporal step of simulation. However,

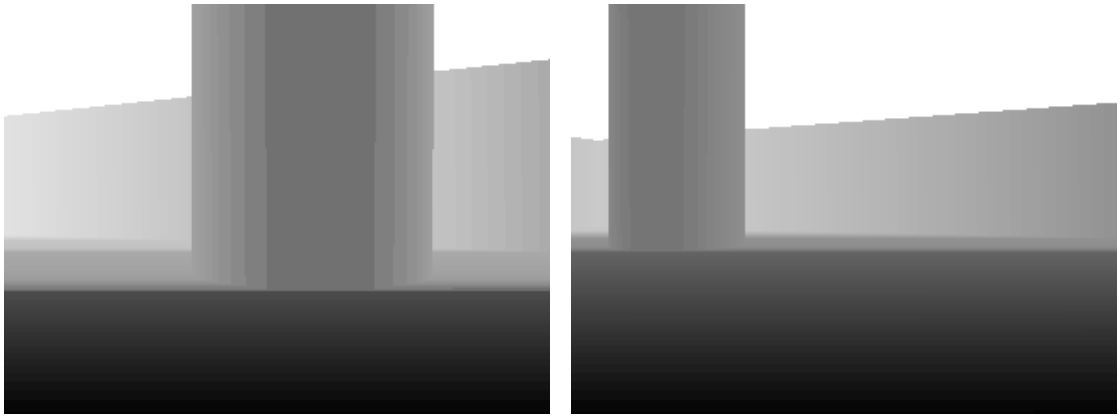

Figure 6.8: Depth images examples during simulation. A column obstacle is captured. Darker pixels indicate smaller distances while lighter pixels represent distant points.

the state s_t and the next state s_{t+1} now contain depth images. Hence, two images are stored as tensors at each time step. This results to be prohibitive for the hardware resources available, also using the RTX 2080 Super GPU with a dedicated memory of 8GB. In addition to that, the dimensionality of the data has largely been increased with respect to the array of LiDAR points. This may constitute a great gap of difficulty in the convergence of the algorithm. For these reasons, a suitable processing of the raw images is necessary to allow for a feasible training in simulation.

The **processing** of raw depth images is composed of mainly three steps:

1. The resolution is reduced from 240x320 to 60x80.
2. A *cutoff* distance value d_{cutoff} is chosen and all the pixels with an higher value are set equal to this threshold value.
3. All the pixels values are normalized with respect to the cutoff d_{cutoff}.

The cutoff d_{cutoff} is fixed to 5 meters to let the network focus on relevant information: closer objects. In such a way all the depth values are bounded in the numerical range [0,1]. For a coherent numerical interval in the input layers of the neural networks, also the goal distance is scaled by the same factor, whilst the goal angle is divided by π. Examples of depth images collected during the simulation with the waffle turtlebot are shown in Figure 6.8. For a better visualization of the scene in those pictures, pixels have been scaled from the normalized range [0,1] to the range [0,255].

6.4.2 Actor-Critic neural networks

The actor neural network is used also in this implementation to control the robot motion. An effective new structure for the actor network is the central challenge of this work. The main idea is to extract features from depth images thanks to convolutional layers. Beside this, it has to be considered that in this task we need to handle input data of different shapes: the depth images (with tensor size [60,80]) and the information about the goal ([1,2]).

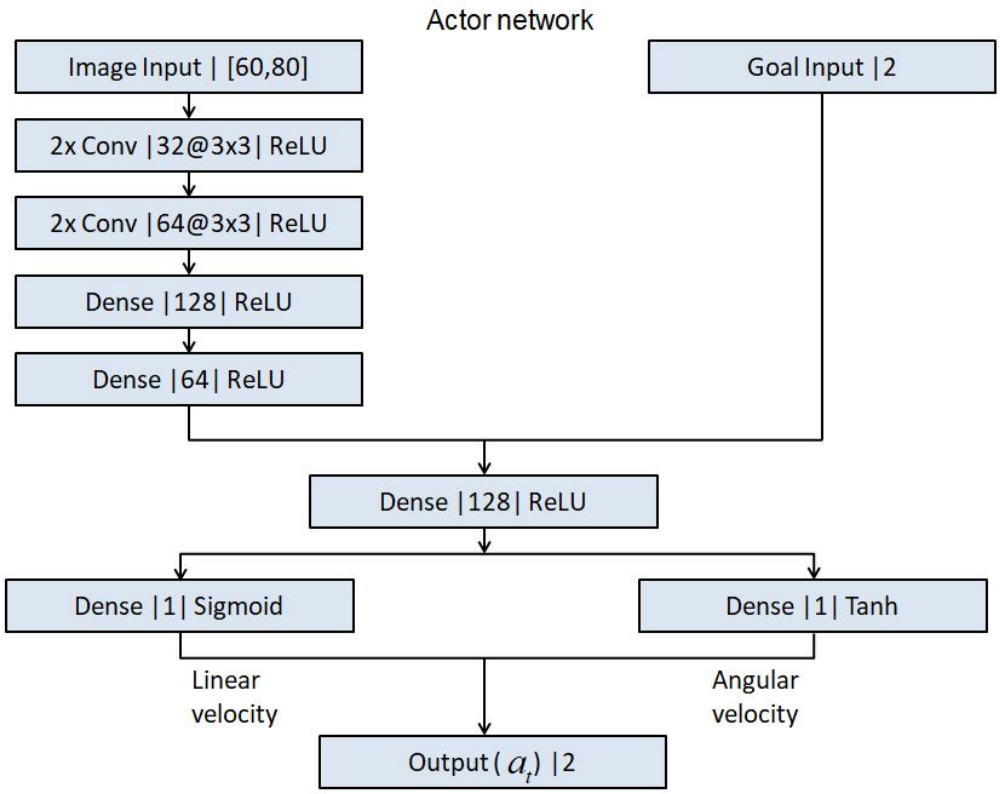

Figure 6.9: Actor neural networks for the visual based navigation system. Images are processed through convolutional layers and then features are aggregated with the information about the goal.

To combine all the data in the input layer, two approaches are thought and experimented. In the first one, images and goal info are passed to the network as two separate inputs. Depth images are forwarded through convolutional layers and the resulting array of features is concatenated with the goal distance and angle. In the second approach, inspired by the AlphaGO project, goal distance and heading are transformed into two constant tensors of the same shape of the image. All of them are stack together in a unique tensor of shape [60,80,3] and this becomes the

unique input of the network. Although it is considered a reasonable method for this task, the first approach results to be more simple and effective. The neural network used for the actor is therefore shown in Figure 6.9.

Four convolutional layers are used to extract features. A limited number of features is considered suitable for this application, since the input tensor volume to squeeze is not huge. The first two convolutional layers have 32 features and 3x3 filters. Then, a max pooling layer (2x2) is applied before two other convolutional layers with 64 features and 3x3 filters. Finally, a global average pool is performed and the obtained array of features is passed to two fully-connected layers before being concatenated with goal information. A single fully-connected layer with 128 neurons is set before the output layer. The sigmoid and hyperbolic tangent activation functions are still used to predict the linear and the angular velocities of the robot.

Critic network follows the same general structure of the actor, with the usual additional input branch for the actions, necessary to predict the Q-value (see Figure 6.10).

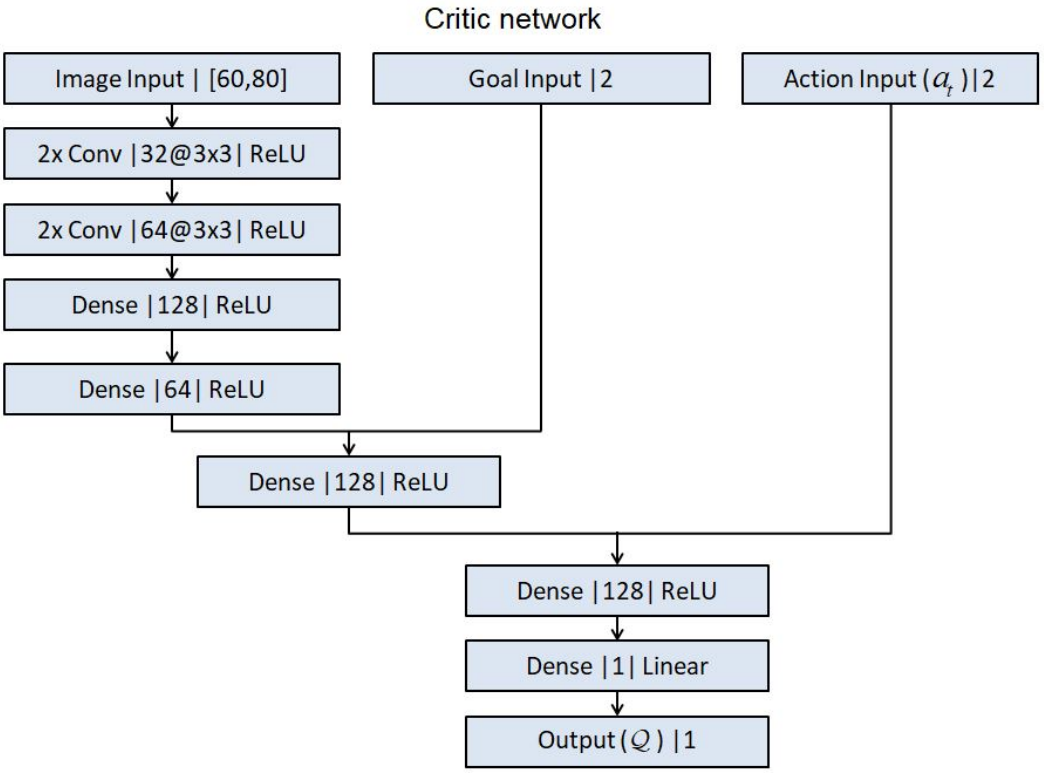

Figure 6.10: Critic neural network for the visual based navigation system.

Beyond the achievement of the task, the design of the convolutional neural networks has been focused also on optimizing the computational cost of the system. In fact, with respect to the LiDAR-based navigation, here the number and the size of fully connected layers have been significantly reduced.

6.4.3 Reward function and training

The same reward function which demonstrates to be effective with many LiDAR points is used for the visual based navigation. A further modification is also applied to the heading sparse reward contribution to improve the obstacle avoidance behaviour.

$$r = \begin{cases} r_g = 100 & \textit{if Goal} \\ r_c = -10 & \textit{if Collision} \\ r_b = k(r_h) + 30(r_{d2}) & \textit{otherwise} \end{cases}$$

Where k is a numerical coefficient decreased with a fixed policy along the simulation. This choice is motivated by the fact that a first stable convergence behaviour without obstacles in proximity of the goal is reached around episode 400 with k = 0.8. Then, when obstacles are gradually added in the virtual scenario, the robot should learn to circumnavigate them in order to reach the goal instead of moving straight toward it. Hence k is reduced and set equal to 0.4 for episodes between 400 and 1000, and then reduced again to 0.2 for the last series of training episodes. Moreover r_g is increased to +200 when the obstacle population becomes dense, after episode 1000. In this final scenario the number of collisions when moving towards the goal may be high and the agent needs a bigger reward.

A totally different approach based on velocity is also tried for sparse rewards, to let the robot learn how to navigate smoothly towards the goal.

$$r_b = k_v(v) - |\omega| + 30(r_{d2})$$

where v and ω are the linear and angular velocities selected in the previous temporal step. k_v is a numerical coefficient, different values such as 2,3,4 have been tried. However, results are not considered satisfying. The scores obtained during the learning process with both the reward approaches are shown in Figure 6.12 and 6.13.

The neural networks are trained in a virtual stage where obstacles are gradually added around. The final configuration of the stage is shown in Figure 6.11.

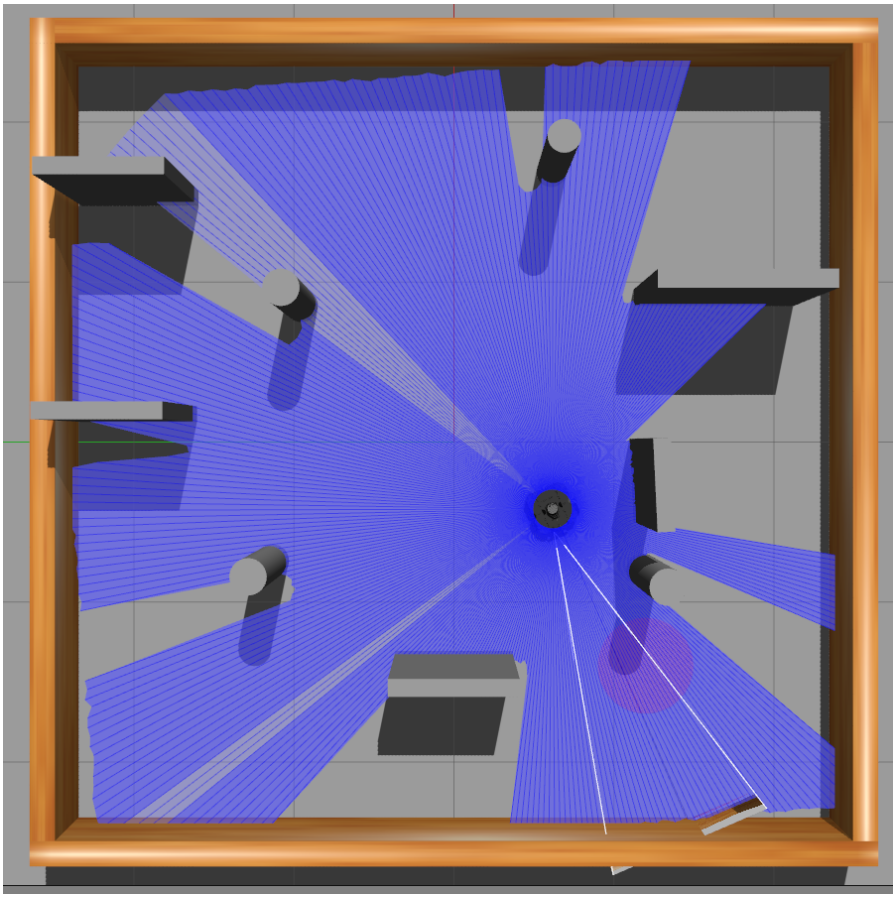

Figure 6.11: Final configuration of the virtual world with static obstacles of different shapes used to train the DRL agent for the visual based navigation.

As already discussed in the LiDAR-based navigation, the neural networks start to be trained after episode 64 and target networks are softly updated from episode 128. ADAM is still the optimizer used for both actor and critic and learning rates are kept the same. The replay memory buffer needs to be limited to a maximum length of 180000 transitions stored. Nonetheless this configuration allow the agent to be sufficiently trained on each phase of the interaction with the environment and it guarantees a working training process. The complete set of parameters used to train the model in simulation is listed in table 6.2.

Parameter	Value
Training hyperparameters	
batch size	64
train start	64
target networks update start	128
replay memory buffer maxlen	180000
actor learning rate	0.0001
critic learning rate	0.0008
tau	0.9
starting epsilon	1
minimum epsilon	0.05
epsilon decay	0.998
discount factor	0.99
Navigation settings	
image resolution	60x80
depth cutoff	$5m$
max linear speed	$0.2m/s$
max angular speed	$1rad/s$
Simulation settings	
max number of episodes	5000
time step	0.001s
max update rate	$1.200s^{-1}$
timeout	500 steps

Table 6.2: Hyperparameters and simulation settings for visual based navigation.

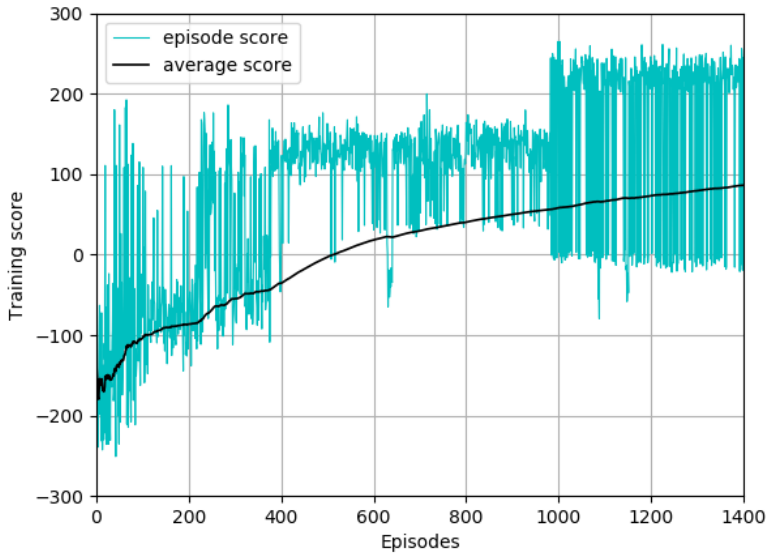

Figure 6.12: Training score trend of a successful simulation with depth images on two different stages. The reward is tuned along the episodes to improve the obstacle avoidance behaviour.

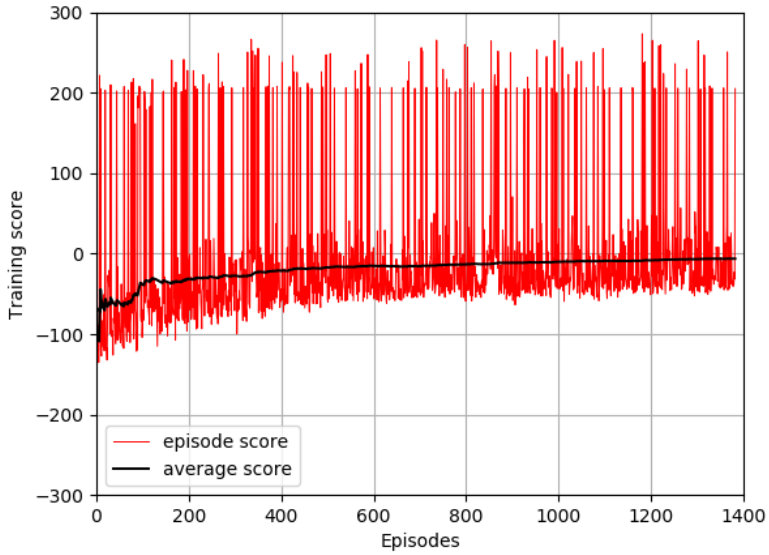

Figure 6.13: Score trend with reward function based on velocity. The plot shows how the speed behaviour is almost satisfied and the agent gets small positive rewards. However the goal is rarely reached.

Chapter 7

Results and Conclusions

7.1 Results

In this last chapter, results obtained from testing the DRL agent for the visual based navigation with depth images are exposed and discussed. A comparison with a LiDAR based agent is also proposed, trying to highlight the different advantages and limitations of the two different solutions.

7.1.1 Metrics

The metrics used to evaluate the performance of the system are mainly focused on the navigation task. A basic set of features is selected for the evaluation:

- **Outcome**: the overall evaluation of the single test. It is used to express if the robot is able or not to reach the specific target point.
- **Total time** $[s]$: the total amount of navigation time spent by the robot, until the single test is terminated.
- **Total path length** $[m]$: the full length of the path chosen by the robot.
- **Final distance from the goal** $[m]$: this metric is only considered in the case of failure.

The set of metrics chosen for the evaluation are merely focused on the final outcome of the navigation test. More accurate metrics concerning the smoothness of the navigation can be used, however, at this point of the project temporal and spatial information about the path are considered sufficient to make some interesting comparisons. A qualitative idea about the quality of the navigation can be also formulated by visually analysing the behaviour of the robot during the tests. Those

considerations are collected in the section dedicated to the qualitative analysis of the results.

7.1.2 Testing simulation

A virtual world for testing is realized in Gazebo (see Figure 7.1). Six different target positions are used to test the ability of the DRL agents in the two configurations developed. For each target, the simulation is reset to the same initial condition. The waffle robot is always spawned at the centre of the stage, where the fixed reference frame for spatial coordinates is located. In this way the virtual world can be fully exploited in all its areas with challenges of different kinds.

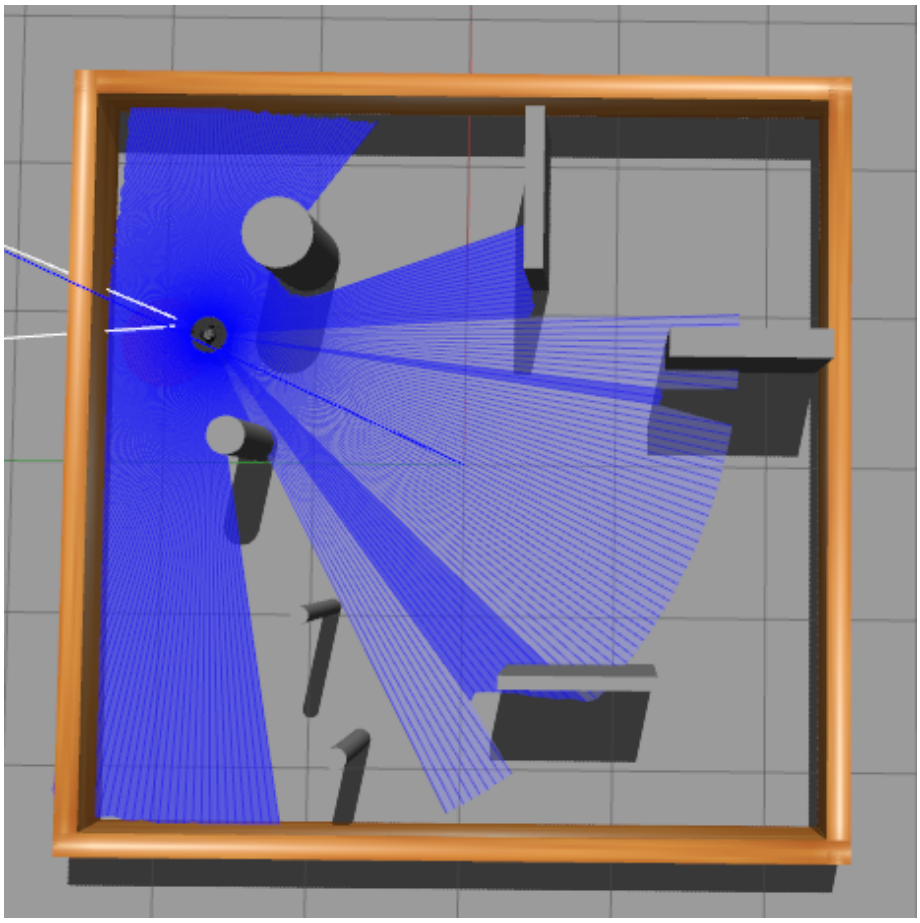

Figure 7.1: Virtual world used for testing. Static obstacles of different shapes and dimensions are placed between the initial spawning point of the robot and the goal position.

Once the target point is set in the virtual environment, the testing episode has three possible outcomes:

- *Goal*: the goal is considered to be reach if the robot gets sufficiently close to it, with a tolerance range $\delta_g < 0.2m$.
- *Collision*: a collision is detected if the distance from an object is lower than $0.15m$.
- *Timeout*: the robot has a maximum of 1000 temporal steps to reach the goal, otherwise the attempt is considered failed. This limit is approximately equivalent to $50s$ of navigation in the simulation.

In Figure 7.2 a sketch of the six targets position in the virtual scenario is provided. Goal 1 is located in the right upper area of the scene and it is used to test the ability to pass through a relatively narrow passage composed of walls. Goal 2, in the same area, requires the robot to navigate close to the wall for a certain path. Goal 3 is located in the upper left corner and the robot has to find a path to avoid the columns in between. Goal 4 is similar with a different disposition. Goal 5 is placed behind thin columns and it represents one of the most interesting challenges to compare the LiDAR with the camera. Goal 6 is at the lower right corner. The robot may choose different paths to reach it.
Then, results are summarized in table 7.1 for LiDAR-based navigation and in table 7.2 for the visual based implementation with depth camera.

	Goal pose	**Result**	**Total time**	**Total path length**	**Final goal distance**
Goal 1	$[1.2, -1.8]m$	Goal	$12.617s$	$2.116m$	/
Goal 2	$[0.2, -2.0]m$	Goal	$12.305s$	$2.004m$	/
Goal 3	$[2.0, 2.0]m$	Goal	$18.536s$	2.751	/
Goal 4	$[0.8, 0.2]m$	Goal	$11.299s$	$2.087m$	/
Goal 5	$[-1.9, 1.2]m$	Collision	$10.463s$	$2.226m$	$1.163m$
Goal 6	$[-2.0, -2.0]m$	Goal	$24.020s$	$3.174m$	/

Table 7.1: LiDAR-based navigation: results obtained from the testing phase.

7.1.3 Qualitative analysis of results

Both the DRL agents demonstrate the ability to reach the majority of the goals, failing once due to a collision with an obstacle. Hence, a first conclusion is that the agents have learnt how to move towards a given target point. By looking at the total

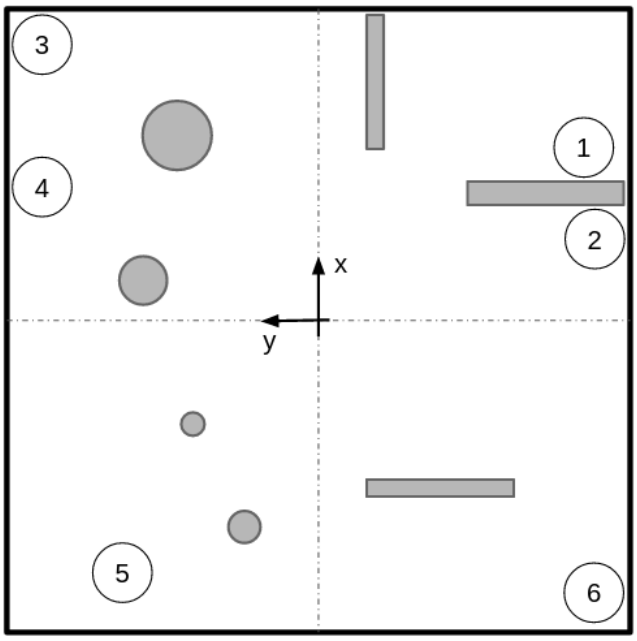

Figure 7.2: Scheme of the virtual world used for testing. Obstacles and goals are sketched for a better visualization of the scenario.

	Goal pose	**Result**	**Total time**	**Total path length**	**Final goal distance**
Goal 1	$[1.2, -1.8]m$	Goal	$11.378s$	$2.076m$	/
Goal 2	$[0.2, -2.0]m$	Goal	$11.194s$	$1.947m$	/
Goal 3	$[2.0, 2.0]m$	Goal	$17.041s$	$2.761m$	/
Goal 4	$[0.8, 0.2]m$	Collision	$8.181s$	$1.389m$	$0.862m$
Goal 5	$[-1.9, 1.2]m$	Goal	$13.743s$	$2.226m$	/
Goal 6	$[-2.0, -2.0]m$	Goal	$22.948s$	$2.801m$	/

Table 7.2: visual based navigation: results obtained from the testing phase.

time and path length spent to reach the goal, lower values can be observed for the camera based implementation. It visually shows a smoother navigation. Angular velocities are chosen such that the robot heading is not affected by oscillations when advancing towards the goal.

The collision events must be analysed critically. LiDAR based agent collide while moving towards goal 5, which is located behind narrow columns. Such a case is clearly a limitation of the LiDAR data, especially when using a reduced number of

measurements. Narrow obstacles are not always detected properly by the sensor rays. It is interesting to see that the visual based agent does not suffer of such a problem. Differently, it is affected by another issue. In fact, the camera-driven robot collides while navigating towards goal 4. By visually analysing its behaviour, it is possible to notice that the robot chooses to first advance along the x axe and only in a second moment it decides to turn left. When the column enters in the field of view of the camera, it is already too close and the robot is not able to counteract and to avoid it. A limited view of the scene is the greatest limit of the navigation based on camera.

A further consideration can be done by looking at the time of navigation obtained for goal 6. Beside the fact that the total distance to cover is greater with respect to other target points, the significant time gap is also related to a particular behaviour of the robot. Indeed, during training the agents autonomously learnt to slow down when more obstacles such as wide walls are getting closer. For this reason, they spend more time for reaching the goal. Although this is not always required, it can be considered a positive behaviour until the goal is reached safely in an acceptable time.

7.2 Conclusions and future works

The DRL agent trained in simulation has shown the ability to reach target points in the presence of static obstacles. The peculiar advantages and limitations of the autonomous navigation system when based on LiDAR points and depth images emerge in the tests. In both cases, obstacle avoidance can be improved working on several aspects. Among them, a training procedure and a reward function more focused on a collision-free navigation are the principal ones. Despite the difficulty of a task such as autonomous navigation with a single depth camera, the project can be considered a precious source of experience and it paves the way for further developments. Future works could be devoted to the following improvements:

- Input data of the neural network can be enriched providing a stack of depth images at successive time instants. This may improve the awareness of the agent about the surrounding scene.
- An hybrid solution based on both LiDAR and depth images can be developed to take advantage of the information provided by the two sensor data.
- The algorithm can be deployed and tested on a physical robot platform. Depth images may need to be corrupted with some noisy signals during training in simulation to adapt them to real sensor data.

Bibliography

[1] Lei Tai, Giuseppe Paolo, and Ming Liu. *Virtual-to-real Deep Reinforcement Learning: Continuous Control of Mobile Robots for Mapless Navigation*. 2017. arXiv: 1703.00420 [cs.RO] (cit. on p. 4).

[2] Amir Ramezani Dooraki and Deok-Jin Lee. «An end-to-end deep reinforcement learning-based intelligent agent capable of autonomous exploration in unknown environments». In: *Sensors* 18.10 (2018), p. 3575 (cit. on p. 4).

[3] Francisco Leiva, Kenzo Lobos-Tsunekawa, and Javier Ruiz-del-Solar. «Collision avoidance for indoor service robots through multimodal deep reinforcement learning». In: *Robot World Cup*. Springer. 2019, pp. 140–153 (cit. on p. 4).

[4] Hao-Tien Lewis Chiang, Aleksandra Faust, Marek Fiser, and Anthony Francis. «Learning navigation behaviors end-to-end with autorl». In: *IEEE Robotics and Automation Letters* 4.2 (2019), pp. 2007–2014 (cit. on p. 4).

[5] Reinis Cimurs, Jin Han Lee, and Il Hong Suh. «Goal-Oriented Obstacle Avoidance with Deep Reinforcement Learning in Continuous Action Space». In: *Electronics* 9.3 (2020), p. 411 (cit. on pp. 4, 6).

[6] Joao Cunha, Eurico Pedrosa, Cristóvao Cruz, António JR Neves, and Nuno Lau. «Using a depth camera for indoor robot localization and navigation». In: *DETI/IEETA-University of Aveiro, Portugal* (2011) (cit. on p. 6).

[7] Daniel Maier, Armin Hornung, and Maren Bennewitz. «Real-Time Navigation in 3D Environments Based on Depth Camera Data». In: Nov. 2012. DOI: 10.1109/HUMANOIDS.2012.6651595 (cit. on p. 6).

[8] *Depth map - Wikipedia*. URL: https://en.wikipedia.org/wiki/Depth_map (cit. on p. 6).

[9] Daniel Seita, Jeff Mahler, Mike Danielczuk, Matthew Matl, and Ken Goldberg. *Drilling down on depth sensing and deep learning*. 2018. URL: https://robohub.org/drilling-down-on-depth-sensing-and-deep-learning/ (cit. on p. 7).

[10] Daniel Seita, Nawid Jamali, Michael Laskey, Ajay Kumar Tanwani, Ron Berenstein, Prakash Baskaran, Soshi Iba, John Canny, and Ken Goldberg. «Deep transfer learning of pick points on fabric for robot bed-making». In: *arXiv preprint arXiv:1809.09810* (2018) (cit. on p. 7).

[11] Ariel Gordon, Hanhan Li, Rico Jonschkowski, and Anelia Angelova. «Depth from videos in the wild: Unsupervised monocular depth learning from unknown cameras». In: *Proceedings of the IEEE International Conference on Computer Vision*. 2019, pp. 8977–8986 (cit. on p. 7).

[12] Zhengqi Li, Tali Dekel, Forrester Cole, Richard Tucker, Noah Snavely, Ce Liu, and William T Freeman. «Learning the depths of moving people by watching frozen people». In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 4521–4530 (cit. on p. 7).

[13] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*. Vol. 1. MIT press Cambridge, 2016 (cit. on pp. 9, 42, 45).

[14] Maurizio Di Paolo Emilio. *Intelligenza artificiale, deep learning e machine learning: quali sono le differenze?* 2018. URL: `https://www.innovationpost.it/2018/02/14/intelligenza-artificiale-deep-learning-e-machine-learning-quali-sono-le-differenze/` (cit. on p. 10).

[15] *Is there a difference between deep learning, machine learning and AI?* 2018. URL: `https://mc.ai/is-there-a-difference-between-deep-learning-machine-learning-and-ai/` (cit. on p. 10).

[16] *ImageNet - Wikipedia*. URL: `https://en.wikipedia.org/wiki/ImageNet#History_of_the_ImageNet_challenge` (cit. on p. 12).

[17] CS231n Convolutional Neural Networks for Visual Recognition. *Biological motivation and connections*. URL: `https://cs231n.github.io/neural-networks-1/` (cit. on pp. 13, 14).

[18] Daniel Graupe. *Principles of artificial neural networks*. Vol. 7. World Scientific, 2013 (cit. on p. 13).

[19] Ž. Ivezić, A.J. Connolly, J.T. Vanderplas, and A. Gray. *Statistics, Data Mining and Machine Learning in Astronomy*. Princeton, NJ: Princeton University Press, 2014 (cit. on p. 18).

[20] Mattew Huttson. *AI researchers allege that machine learning is alchemy*. 2018. URL: `https://www.sciencemag.org/news/2018/05/ai-researchers-allege-machine-learning-alchemy` (cit. on p. 24).

[21] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: `1412.6980 [cs.LG]` (cit. on p. 34).

[22] Lilian Weng. *A (Long) Peek into Reinforcement Learning*. 2018. URL: https://lilianweng.github.io/lil-log/2018/02/19/a-long-peek-into-reinforcement-learning.html (cit. on p. 38).

[23] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG] (cit. on p. 51).

[24] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. *Continuous control with deep reinforcement learning*. 2015. arXiv: 1509.02971 [cs.LG] (cit. on pp. 54, 55).

[25] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. «Deterministic Policy Gradient Algorithms». In: *31st International Conference on Machine Learning, ICML 2014* 1 (June 2014) (cit. on p. 55).

[26] *Turtlebot3*. URL: https://emanual.robotis.com/docs/en/platform/turtlebot3/specifications/#specifications (cit. on pp. 63, 65).

[27] *ROBOTIS dynamixel actuators*. URL: http://en.robotis.com/model/page.php?co_id=actuator (cit. on p. 64).

[28] *Turtlebot3 RealSense*. URL: https://emanual.robotis.com/docs/en/platform/turtlebot3/appendix_realsense/ (cit. on p. 66).

[29] Enrico Sutera. «Deep Reinforcement Learning and Ultra-Wideband for autonomous navigation in service robotic applications». MA thesis. Politecnico di Torino, 2019.

[30] Anna Boschi. «Person tracking methodologies and algorithms in service robotic applications». MA thesis. Politecnico di Torino, 2019.