Politecnico di Torino

Master of Science Degree in MECHATRONIC ENGINEERING

Master Thesis

Deep Reinforcement Learning and Ultra-Wideband for autonomous navigation in service robotic applications

Supervisor:
prof. Marcello Chiaberge

Candidate:
Enrico Sutera
S253364

December 2019

Abstract

Autonomous navigation for service robotics is one the greatest challenges and there's a huge effort from scientific community. This work is born at PIC4SeR (PoliTo Interdepartmental Centre for Service Robotics) with the idea of facing the aforementioned challenge merging rediscovered and promising technologies and techniques: Deep Reinforcement Learning and Ultra-Wideband technology.

Over few past years the world has seen a huge advance in the field of Artificial Intelligence, especially thanks to Machine Learning techniques. The latter include a branch called Deep Reinforcement Learning (DRL) that involves the training of Artificial Neural Network (ANN) from experience, i.e. without the need of huge datasets. Here DRL has been used to train an agent able to perform goal reaching and obstacle avoidance.
Ultra-wideband (UWB) is an emerging technology that can be used for short-range data transmission and localization. It can be used in GPS-denied environments, such as indoor ones. In this work UWB has been used for localization purposes. UWB is supposed to be a key technology in future: many giant companies are involved and Apple has already inserted an UWB chip in its latest product.

It has been used a differential drive robot as implementation platform. The robot is controlled by an ANN (which has robot pose information, lidar information and goal information as input and linear and angular speeds as outputs) using ROS (Robot Operating System). The ANN is trained using a DRL algorithm called Deep Deterministic Policy Gradient (DDPG) in a simulated environment. The UWB has been used in testing phase only.
The overall system has been tested in a real environment and compared with human performances, showing that it is able - in some tasks - to match or even outdo them. There have been satisfying results and it is believed that, although there are strong limitations given by the difficulty of the challenge, the system complies with expectations and constitutes a good baseline for future work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Objective of the thesis

Autonomous navigation for service robotics is one the greatest challenges and there's a huge effort from scientific community, because fully autonomous since could really improve humans life. At PIC4SeR (PoliTo Interdepartmental Centre for Service Robotics) it has been decided to face the aforementioned challenge merging rediscovered and promising technologies and techniques: Deep Reinforcement Learning and Ultra-Wideband technology.
In the few past years the world has seen a huge advance in the field of Artificial Intelligence, especially thanks to Machine Learning techniques. The latter include a branch called Deep Reinforcement Learning (DRL) that involves the training of Artificial Neural Network (ANN) from experience, i.e. without the need of huge datasets. Here DRL has been used to train an agent able to perform goal reaching and obstacle avoidance. Ultra-wideband (UWB) is an emerging technology that can be used for short-range data transmission and localization. It can be used in GPS-denied environments, such as indoor ones. In this work UWB has been used for localization purposes. UWB is supposed to be a key technology in future: many giant companies are involved and Apple has already inserted an UWB chip in its latest product.

The main argument of this thesis work is hence to develop and implement such methodologies and algorithms to achieve a system able to perform navigation from one point to another not only autonomously and map-less but also without the need for huge hardware requirements.

It is used a differential drive robot as implementation platform. The robot is controlled by an ANN (which has robot pose information, lidar information and goal information as input and linear and angular speeds as outputs) using ROS (Robot Operating System). The ANN is trained using a DRL algorithm called Deep Deterministic Policy Gradient (DDPG) in a simulated environment. The overall system is tested in a real environment and compared with human performances.

1.2 Organization of the thesis

This thesis consists of eight chapters and it's organized as follows:

In chapter 1 an overview of the work and its motivations are provided. Moreover a brief description of this dissertation is given.

In chapter 2 the navigation problem is shortly tackled. Next the state of the art in autonomous navigation in service robotics with the use of Deep reinforcement Learning is taken into account.

Chapter 3 provides a description of Deep learning, starting from its history till main used - for this dissertation purposes - concepts and elements.

Chapter 4 takes into account Reinforcement Learning theory and algorithms, which are then implemented in chapter 6.

Chapter 5 is dedicated to Ultra-wideband technology, whose principles are basically explained.

In chapter 6 all the practical work is explained. First the hardware used is taken into account, hence both the robot and the Ultra-wideband sensors. Then an overview of software tools is provided. Finally the overall implementation is taken into account, along with platform and algorithms settings.

Chapter 7 is the last one and it contains the results of the work. First a brief description of the metrics is given, next the test set-up is explained, followed by results and conclusions.

Chapter 2

State of the art

2.1 Introduction

Navigation problem (known also as motion planning) in robotics, refers to the achievement of a destination from a source, by finding a sequence of valid configurations of the robots.
A trivial and extremely fitting example can be a mobile robot in an indoor environment partially filled with obstacles. Till few years ago this problem has been faced using those that now can be referred to as "traditional algorithms". Traditionally the navigation problem is faced using a complex system made of many elements. Briefly, a robot, in order to reach any target needs:

- a *localization system* which has to provide a pose with respect to a reference frame;
- a *path planner* whose aim is to compute a sequence of valid configurations that can bring the robot from the starting point to the target;
- a *navigator* that receives localization information and tries to follow the sequence provided by the path planner, by controlling the robot.

Basically each of them keeps working during all the task and generally the related algorithms are not much simple or computationally light. Hence there's the need to find more efficient algorithms. Moreover traditional ones have the disadvantage to be following rules that are given by the programmers; this of course give rise to limitations and lack of autonomy. The idea of having an autonomous navigation does need a system able to generalize and that can somehow face situation that have not been taken into account during the design phase. In this sense, DRL can be useful both for giving flexibility, to decrease computation requirements. Furthermore in DRL paradigm path planner and navigator are merged.

2.2 Autonomous navigation and Deep-RL

Deep reinforcement learning has been used in this field only recently. From 2015 there has been a lot of work, and many results have been achieved. Map-less navigation has already been faced using DRL and obtaining good results. For further information concerning what follows sources can be seen [1],[2],[3],[4],[5].

"Semi-supervised Deep Reinforcement Learning in Support of IoT and Smart City Services"

This work was published in 2017. Concerning a smart city contest, Iot devices have been used to provide partially labelled data, which then could be use for training purposes. The work proposed a semi-supervised DRL model that is able to increase its accuracy. The DRL agent has to reach a target and hence the field is mobile service robotics. The novelty has been the introduction of these partially labelled data that are said to allow an improvement with respect to standard supervised DRL models.

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

This work was published in 2017. The authors provided a map-less learning-based system able to perform goal reaching and obstacle avoidance. The system make a representation of the state thanks to 10 frontal measures of lidar (it hence is not aware of what's behind), to its position and velocities, and to target position. The results are compared in terms of path optimality and computational request, showing that DRL-based systems can be effective.

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

This work was published in 2018. It presented a robot controlled by a MDLR (Memory-based Deep Reinforcement Learning) which could provide a high degree of autonomy in a unknown environment. The training of the net has been performed in simulation, while the whole system has been tested in a real environment. The work shows that the presented algorithm can learn autonomously and continuously.

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

This work was published in 2019. It proposes and end-to-end approach for indoor navigation of service robots using Deep Reinforcement Learning (DRL). The algorithm

used for the training is a DDPG. The controller provides continuous actions as output, while receiving information from a camera (depth image), odometry and laser.

"Fully Distributed Multi-Robot Collision Avoidance via Deep Reinforcement Learning for Safe and Efficient Navigation in Complex Scenarios"

This work was published in 2018. The authors developed a collision avoidance policy for a multi-robot system based on DRL. This approach is robust and not heavy in terms of computational demand in acting phase. The main advantage is that there is no interconnection between robots, hence the system robustness is higher and safety does not depend on communication. This multi robot system has been tested in simulation with 100 robots, while in a real environment with a still significant number of agents.

2.3 Ultra-WideBand today

Ultra-wideband has already been used in localization application, for instance, in 2019 range measurements were used to provide information about the position of mobile robot [6]. In 2011 a comparative study was done between UWB localization system and SLAM (Simultaneous Localization and Mapping) algorithm, showing that UWB can provide valuable real-time information [7].

Chapter 3

Machine Learning

3.1 What's in this chapter

The chapter is aimed to give to the reader the concept behind machine learning (DRL). To do so it's convenient to face some simpler sections. First there's a brief overview on history of machine learning, then its basis are taken into account, in particular to understand what a neural net is and how it works.

3.1.1 AI, machine learning, deep learning

Nowadays there's some confusion when talking about AI (artificial intelligence), machine learning (ML) and deep learning, because it can be not really clear how they are related each other. This is also due to the fact that they simply are born, as terminology, in really different times: AI is likely to be located in antiquity, ML was first called this way by A. Samuel in 1959 [8] , while DL is a more recent thing. An easy way to think of the relationship between them
is to imagine concentric circles with AI — first idea chronologically — the wider, then machine learning — which came later, and then deep learning — which is leading today's AI interest explosion — fitting inside both. AI is indeed more an idea of something able to act in a smart way, hence it can be related to everything that could give live to and AI. ML is a set of techniques to perform automatic learning, and DL can be considered to be a more specific set of techniques. Though they could be distinguished, in the following there won't be a clear division.

3.2 History of machine learning

The first embryo of neural net can be dated to 1943, with McCulloch-Pitts' early model of human brain. Their neuron could do binary classification of inputs by checking the sign of a function $f(\boldsymbol{x}, \boldsymbol{w})$, where $\boldsymbol{x}$ is the vector of inputs and $\boldsymbol{w}$ is a vector of weights. However, weights had to be correctly set by hand. In the '50 Alan Turing thinks of a

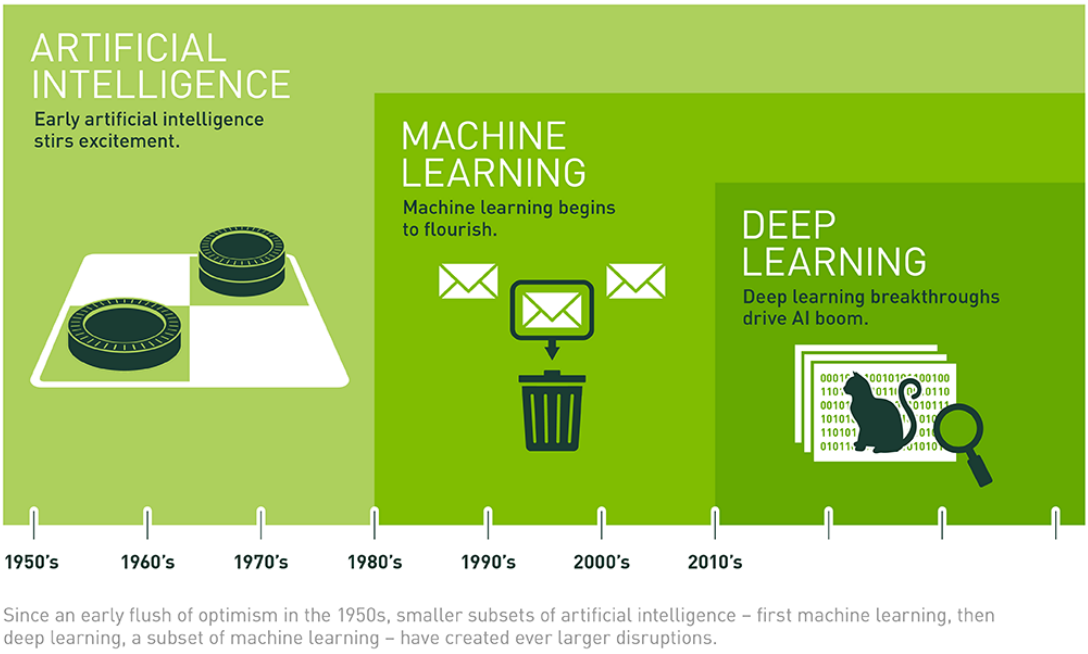


FIGURE 3.1: Artificial intelligence, machine learning and deep learning. [9]

test to check whether a machine could or not fool a person, by making them believe to be talking to a human instead of a machine. That period represents the roots of what has come later. In figure 3.2 a scheme of main events from those times is provided. 1957 is the year of the Perceptron, which represents a huge step ahead in deep learning. Again, Frank Rosenblatt, takes inspiration from human brain, so that the Perceptron resemble human neurons. In contrast to McCulloch-Pitts's, the Perceptron is able to learn through an iterative process, by comparing outputs and inputs and modifying its weights. Hence it was a versatile binary classifier. Rosenblatt idea had a huge success in the following years because there were a lot of task it could fit. However, some years later (1969), Marvin Minsky and Seymour Papert claimed that due to the intrinsic linearity of the Perceptron it could not solve problems such as XOR one. This led to the period known as "the first winter" of AI.
Lately, in the 1980s, another wave of research on neural networks arose, especially via a movement called **connectionism**. The idea was that a great number of simple units could achieve an intelligent behaviour when working together, as a net. This led to introduction of hidden layers in neural nets, thanks to Rumelhart and Williams and the back-propagation algorithms. In the 1990s the long short-term memory (LSTM) network were introduced by Hochreiter and Schmidhuber to solve some important difficulties in modelling mathematically long sequences. This second wave of research lasted till the mid-1990s. Then, until 2007 neural networks were believed to be very difficult to train and way too much ambitious.

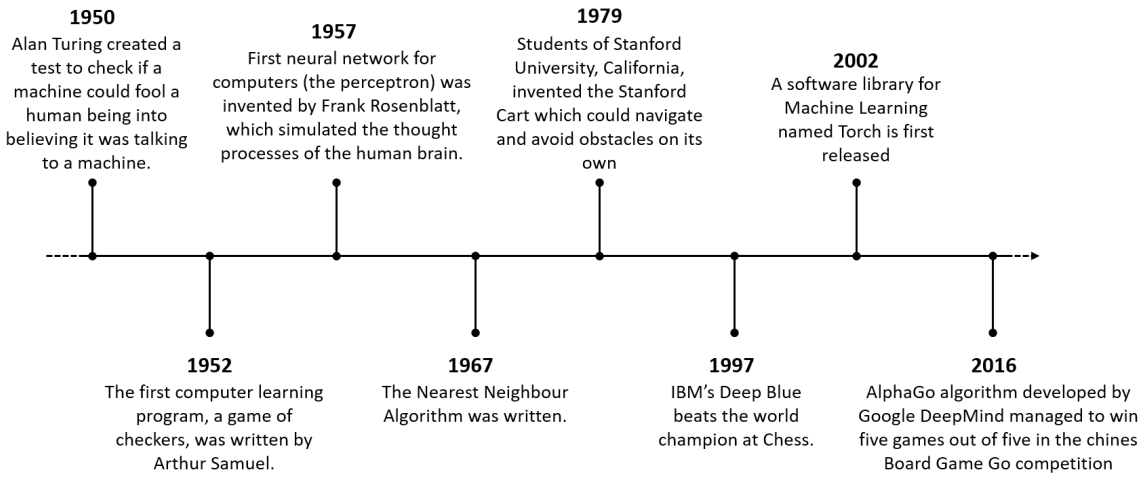


FIGURE 3.2: Time-line concerning Deep Learning history.

However, in this period there were also impressive result, e.g. thanks to Yoshua Bengio, Yann LeCun, Geoffrey Hinton who led different programs of research on machine learning in different universities. From 2006 it was understood that deep (neural) networks could be trained. At that time deep learning outperformed AI systems based on other ML techniques.

However, only recently deep learning has been identified as a crucial technology, though now it's clear it's development process has started back in the 1950s. One of the reason is that today we can provide the right resources to these algorithms.

First, as the world have become more digitalized and inter-connected we have reached the so called age of "Big Data". With this comes a huge increase in the size of available dataset. During the already cited second winter, one of the problem was the restricted size of dataset: it was often quite hard for a net to generalize. To understand the scale of this, it's sufficient to know that at the beginning of 1900s biggest datasets concerned criminals and had a size lower than 10^4; nowadays there are datasets with a size order even five times greater.

Moreover, recently there has been a huge increase in computational resources - especially for faster CPUs and the spreading of general purpose GPUs - that allowed the running of much larger models. One thing that should be clear is that a single neuron or few of them are not really useful. Indeed, with the number of biological neurons the observable intelligence increases (e.g. human beings have about 10^11 neurons, octopus slightly above 10^8 and roundworms just some hundreds). Today, with biggest nets we settle on the 10^6 neurons.

Another difference with respect to previous years is the number of connection per neuron. For a human neuron it is about 10^4, not an exorbitant number; indeed it had already been achieved back in 2013.

3.3 Machine learning concepts

In the followings most important concept will be briefly introduced, to let the reader understand how a neural network works, starting from the simple unit.

3.3.1 Threshold logic unit - TLU

As already discussed in the previous pages, the central idea from which neural networks are born, is the biological neuron. Hence, in the following a short phenomenological and morphological description of it is given. In figure 3.3 a neuron cell body representation is shown, among other two units of the same type.

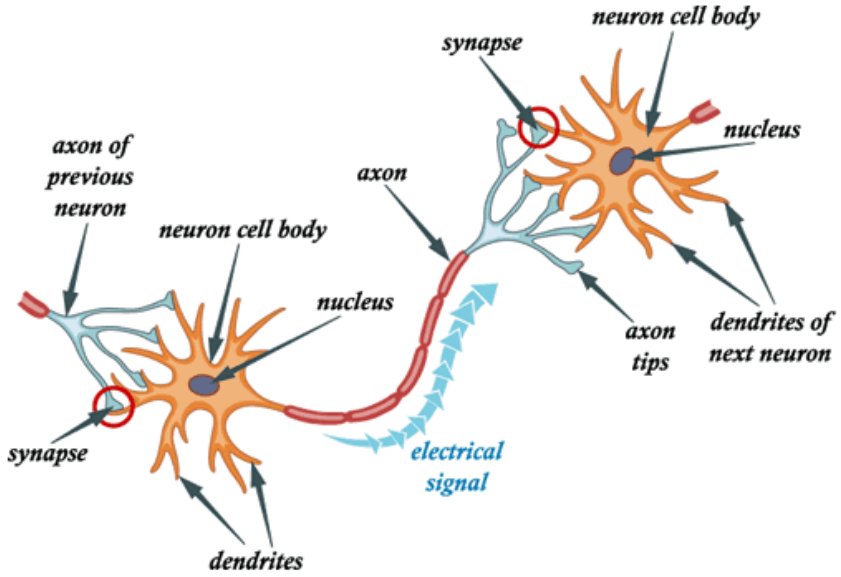


FIGURE 3.3: Biology scheme of a neuron([10]).

A neuron is composed by some main elements [11]

- Neuron cell body : it contains the core of the cell, the nucleus.
- Dendrites : they are extensions that start from the body. They work as receiver and transmit information to the cell body;
- Axon : it's another extension that start from the body, but it's unique and works as a channel for transmission to the synapses;
- Synapses : it's a set of ramifications that are born from the axon. Their role is to transmit information to others neurons.

To understand how a neuron works, let's imagine that there's one transmitting a signal (it is actually not electrical conduction but a voltage-gated ion exchange by means of electro-chemical process along the axon) to its synapses through its axon. When the

signal is near the synapses, these release an amount of *neurotransmitters*, a chemical substance. This quantity, released in the synaptic gap determines the synapse *conductivity*, that can be seen as a measure of how much it attenuates or boots the signal received from the axon. Downstream of the synaptic gap, the post-synaptic neuron is provided with receptors, called *dendrites*, able to catch neurotransmitters. Next, local small current are generated close to the synapses and they can sum up in time and space. If this value is greater than a certain threshold, it is generated an impulse with some entity and duration than is transmitted again through the axon, so that the whole process is repeated. Artificial neural networks are the result of reverse engineering of the just introduced biological structure. Indeed, artificial neurons are no more than *units* able to receive inputs and eventually fire an output. The threshold logic unit (TLU) is a simple model of the biological neuron, and a parallelism can be easily done. The signal that goes through the axon is now represented by a number, usually between 0 and 1. The amount of neurotransmitters released, that is the *conductivity of the synapses*, is now associated to the *weights of the connection*(w_j). The signal is fired depending on a certain activation function, that is, a certain threshold (see figure 3.4).

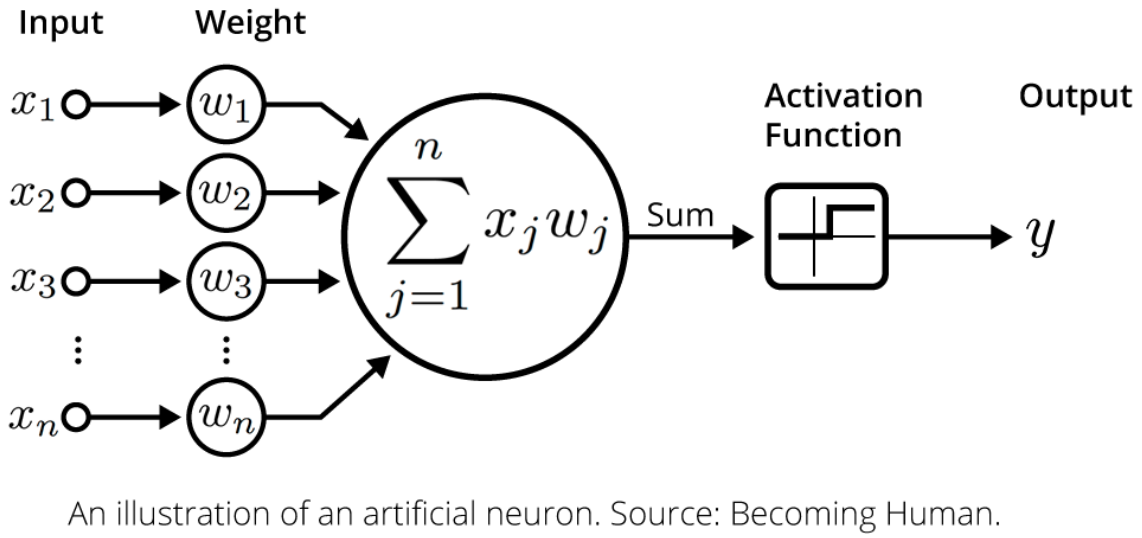


FIGURE 3.4: Scheme of the TLU.

There are some rules that the model has to follow:

- artificial neurons have a given threshold, θ;
- logic units have a binary output;
- neurons have a link with an inhibitory signal;
- any neuron receive some input signals and they all have identical weights;
- if no inhibitory signals are present, the all the weighted signals are summed and the output signal may be 1 if it's greater than the threshold or 0 if not.

The behaviour of the model can be expressed as follows:

$$f(n) = \begin{cases} 1 : \sum_{j=1}^{n} w_j x_j \geq \theta \wedge no_i nhibition \\ 0 : otherwise \end{cases}$$

3.3.2 The Perceptron

With respect to the TLU the perceptron has some feature:

- Values of weights and biases are not identical;
- Weight values may be both positive or negative;
- There's no inhibitory signal;
- Perceptron comes with a learning rule;

It has an activation function similar to the TLU one

$$f(n) = \begin{cases} 0 : \sum_j w_j x_j \leq threshold \\ 1 : \sum_j w_j x_j > threshold. \end{cases}$$

Considering **w** as an array and b as inverse of the threshold, it can be re-written

$$f(n) = \begin{cases} 0 : w \cdot x + b \leq 0 \\ 1 : w \cdot x + b > 0 \end{cases}$$

in which a is the activation function. The bias can be thought as some kind of threshold that considerably influences the output of the unit. Indeed, by considering and high value of it, it is likely that the output is going to be equal to one.
However, the novelty that comes with it is also the most important: the idea of having a learning algorithm which allows a use of artificial neurons in a new different way, with respect to conventional logic circuits. Indeed, this unit could learn to solve some problems on its own, by adjusting its parameters. It is done a trivial way. Once defined the output as $y = y(x_j)$, with x being the inputs, a the actual output, it's possible to define also the *delta error*. The latter is simply the difference between the desired output and the current one. Hence a correction can be performed:

$$\delta = (y(x_j) - a)$$

$$\Delta w_j = \eta \cdot \delta \cdot a$$

being η the learning rate $(0 \div 1)$. Hence weights are modified in order to raise or decrease the value of the output, to make it match the desired output.

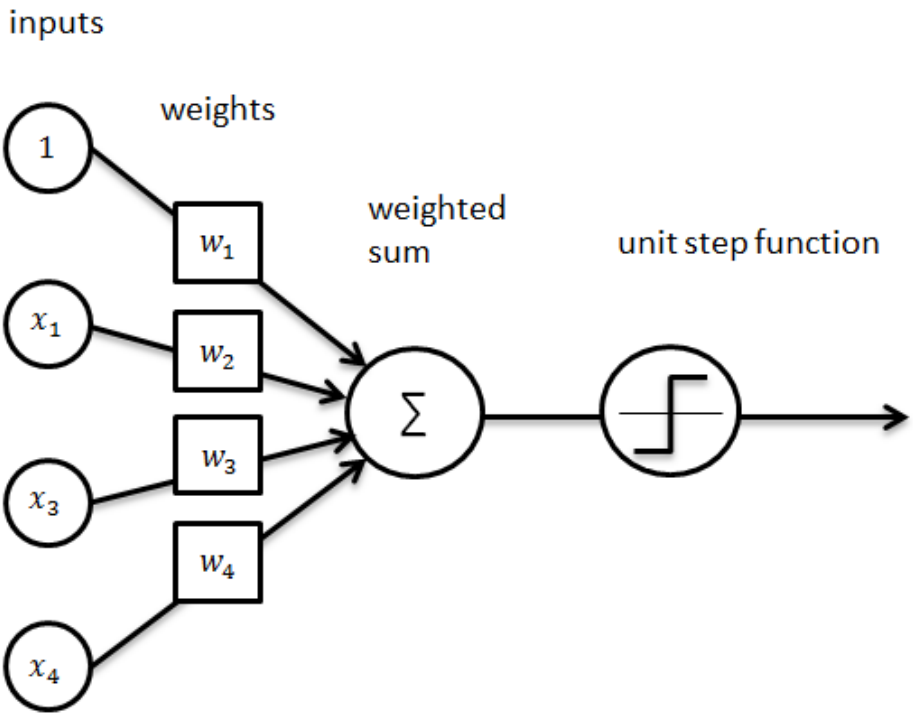


FIGURE 3.5: Scheme of Rosenblatt's perceptron.

3.3.3 Framework of Neural Networks

The one thing in common between the units presented is the single output. However, this structure gives problem when facing problems such as the XOR one, that is, the learning of non-linear functions. To overcome this concern, neural network were subjected to a transformation, in order to add more mutually interconnected layers. The notation using in the following can be understood thanks to figure 3.6.
Since this new architecture has got more weights and biases, the learning algorithm is more complex. The network used as example has four layers. The leftmost layer is made of units known as *input neurons*; together they form the so called *input layer*. On the other side there'ss a layer known as *output layer* that is made by *output neurons* and that in this specific case contains only one unit. The two external layers are connected by the intermediate *hidden layers*. Four parameters are considered necessary and sufficient in order to describe the neural net:

- x_j is the j_{th} input, which is received by the j_{th} input neuron. Generally they are collected in a vector x;
- b_j^l is the j_{th} bias of the l_{th} layer. Bias are collected in a vector too, in this case called b^l, for each layer;

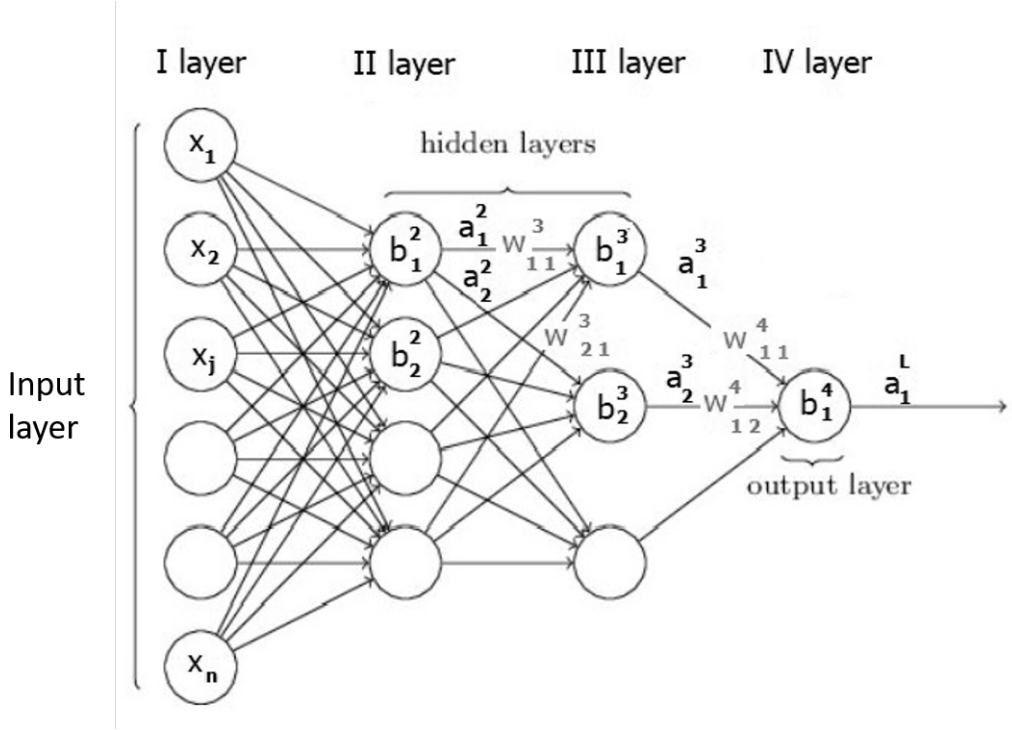


FIGURE 3.6: Four-layer neural network nomenclature.

- a_j^l refers to the activation function of the j_{th} unit in the l_{th} layer. As before, a^l is the related vector;
- w_{jk}^l identifies a weight for the link from the k_{th} neuron in the $(l-1)^{th}$ layer to the j_th unit in the l_th layer. In this case all weights are compressed in a matrix for each layer w^l.

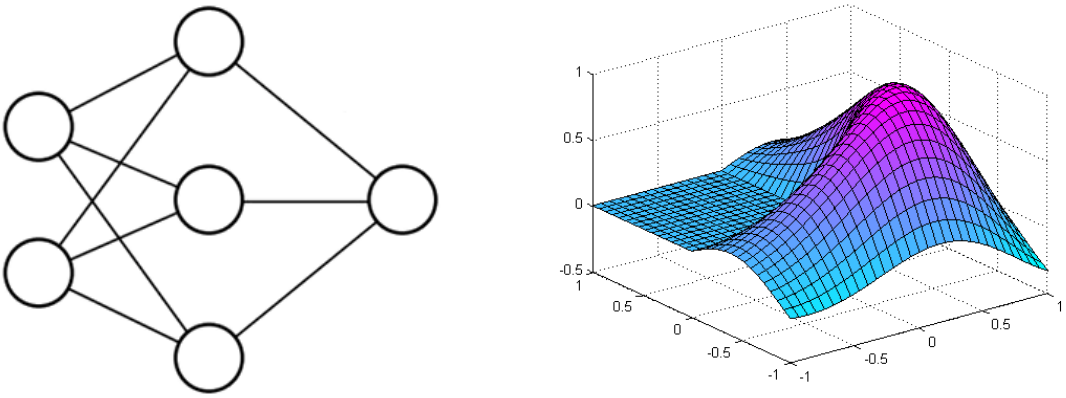
The differences between this structure and the one seen before (the perceptron) can be noticed using the activation function generated at each neuron given by:

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right) \longrightarrow a^l = \sigma\left(w^l a^{l-1} + b^l\right)$$

With one hidden layer only, a network can compute a huge number of functions but with two, its capabilities grow exponentially. It is trivial to understand, since the second hidden layer allows to sum more complex functions that come from the previous layer. As example, adding one hidden layer only, the net manage to pass from the impossibility of learning the aforementioned function XOR to the computation of a much more complex structure as showed in figure 3.7.

3.3.4 Activation functions

In modern neural networks different kinds of activation units are exploited. Many models have been developed over the time. However, the following are most used.



(A) Three layers neural network having two input neurons and one output neuron.

(B) Plot of the possible output graph.

FIGURE 3.7: Visual representation of the computational capabilities of a quite simple neural network.

Sigmoid Unit

The learning procedure involves changes of weights and of biases (the perceptron learning procedure instead implies weights change only). This of course creates differences in the output as presented in figure 3.8.

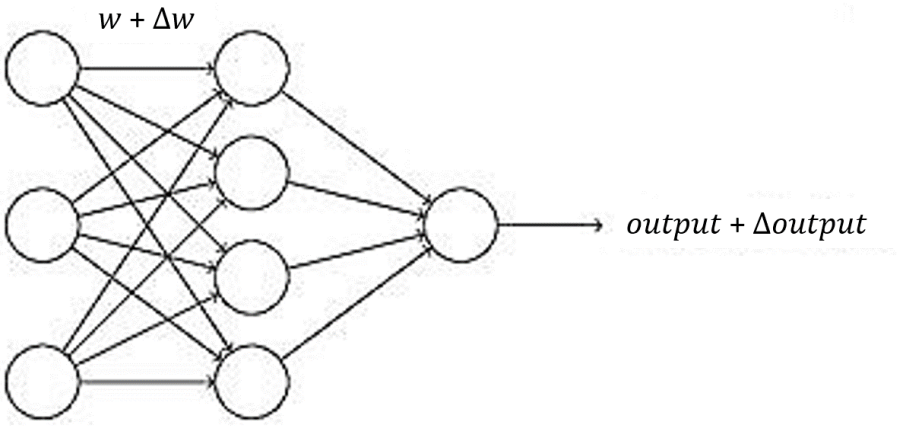
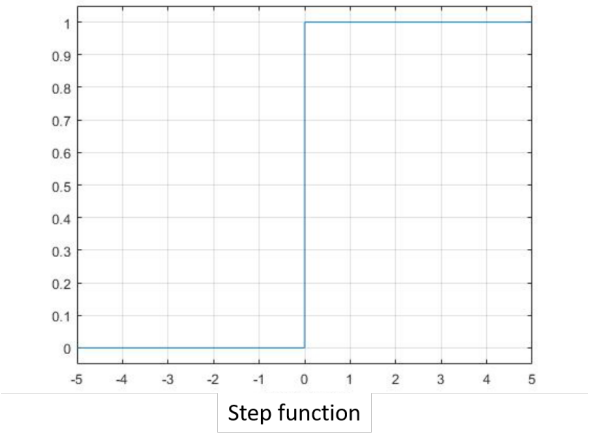
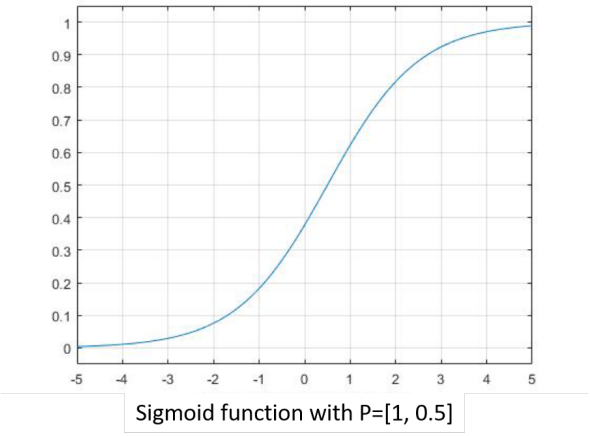


FIGURE 3.8: The weights variation is propagated along all the network neurons resulting in a variation of the output.

Thinking of the network as made again of perceptrons, which due to the expression of their activation functions, have small changes at the input layer of the network that produce a completely different result at the output layer. So, if for instance the output was zero, it could shift to one having just slightly variations of some variables. Hence the introduction of the *sigmoid function* is aimed to lessen this effect of small variations



(A) Perceptron activation function. Output can only vary from zero to one depending on the bias.



(B) Sigmoid function output can have any values between zero and one.

FIGURE 3.9: Step function and sigmoid function curves plots.

and to balance them with respect to the final output.

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

The new activation function result:

$$\sigma(wx + b) = \frac{1}{1 + \exp\left(-\sum_j w_j x_j - b\right)}$$

This new variety of unit overcomes the most important problem of the perceptron. Indeed, its output function doesn't have any discontinuous behaviour, rather it react in a smoother manner to input variations. The sigmoid function can also be seen as a polished perceptron activation function. The weights and biases indeed change the slope and the position of the plot as illustrated in figure 3.10.

Linear Unit

A linear unit has a transfer function that generate an output value equal to the activation potential. It doesn't make any adjustment to the input $a = \sigma(z) = z$. The produced graph is a simply straight line going withstanding first and third quadrants.

Tanh Unit

The hyperbolic tangent (*tanh*) function can be a great alternative to the previous seen sigmoid function. Different papers have proved that this type of artificial neurons perform actually better than other activation units in some situations. The most

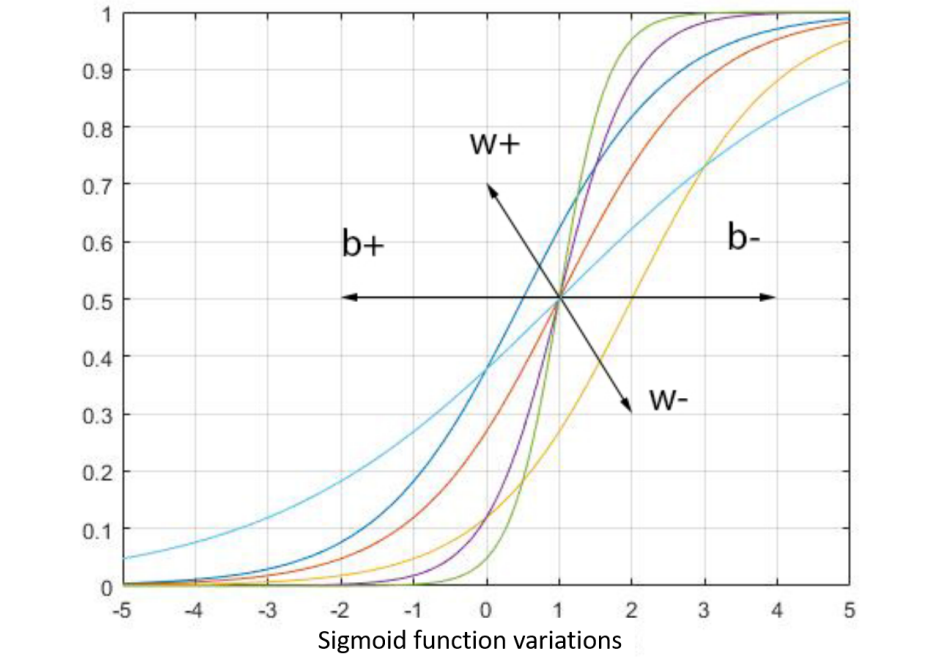


FIGURE 3.10: Depiction of how weights and bias influence the shape of the graph.

remarkable difference with the other ones can be discerned in figure 3.11; indeed, the output has a range bounded by -1 and 1 and not within 0 and 1. This should be kept into account for the specific devised applications. The described function is characterized by the equation:

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

hence its activation function can be computed with $tanh(wx + b)$:

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

Rectified Linear Unit (ReLU)

ReLU is also recognized as a ramp function along with being similar to the well known half-wave rectification of the electrical engineering field. Currently, the rectifier is the most favored activation function, principally for deep neural networks. This is due to a multitude of factors (such as the *vanishing* or *exploding gradient*). The expression of the aforementioned function is :

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

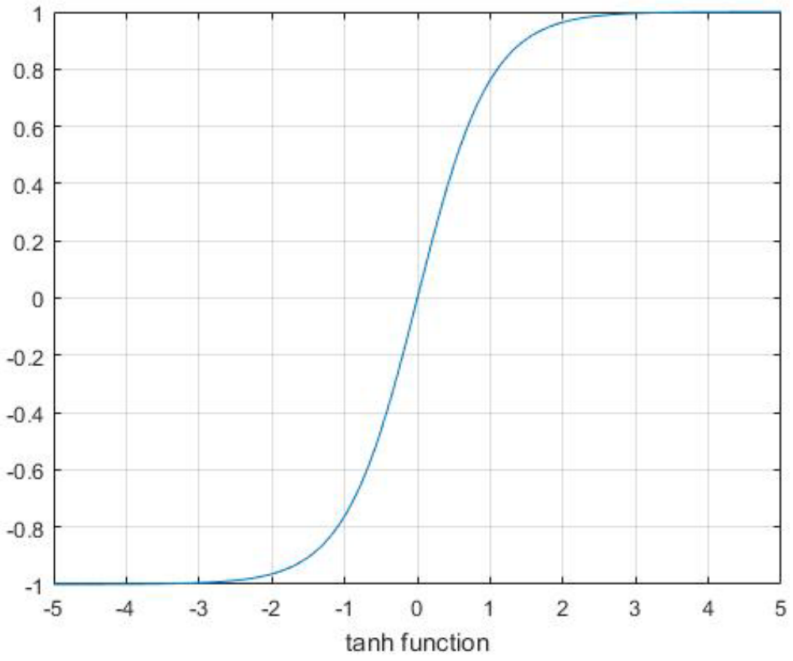


FIGURE 3.11: Plot of the tanh unit activation function. The selection of values in the vertical axis is bounded by -1 and 1.

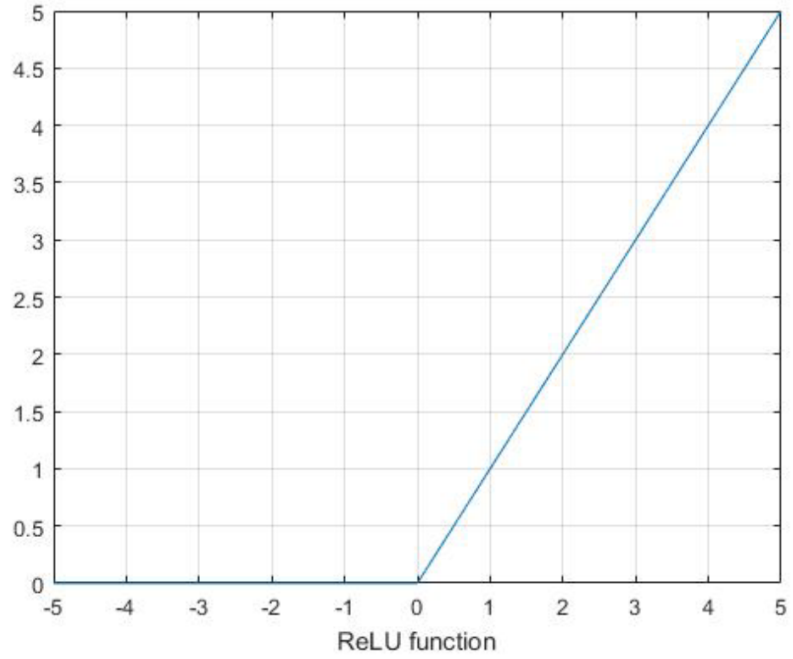


FIGURE 3.12: Plot of the activation function of rectified linear unit (ReLu).

Softmax Unit

These special units are largely used in the output layer of neural network, particularly for problems related to classification. Hence this unit is always adopted in the closing

layer and a^L is a vector made of output factors a_j^L and so $z^L = w^L a^{L-1} + b^L$.

$$a_j^L = \sigma\left(z_j^L\right) = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}}$$

What is really interesting about this unit is that it can be thought as a probability distribution, as:

$$\sum_j a_j^L = \frac{\sum_j e^{z_j^L}}{\sum_k e^{z_k^L}} = 1$$

Thus, for this last aspect, softmax units are more and more used in the output layers. Therefore, it is possible to depict networks predictions as probabilities, or a quick examination of the confidence of the network in its response.

3.3.5 Gradient Descent

Now it is clear how a simple neural network works. More complicated structure require different computation from the essential ones, to learn and to provide significant results. At this purpose is advantageous to introduce one of the central notions in neural networks: the cost function. It is the factor that quantify if the net is adjacent or not to the goals. One of the most known, straightforward and nowadays most exploited is the mean squared error (MSE):

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

where $y(x)$ are the wanted output and a the actual one. An hypothetical impeccable training algorithm should have variables such that $C(w,b) \approx 0$. It means the net should cyclically adapt the variables to minimize the aforementioned cost function. In order to identify this algorithm it is appropriate to consider a generic situation in which the input of function of cost is a n-dimensional array v. It is impractical and in some cases not even unattainable to use calculus to minimize the $C(v)$ function. Another method has to be followed due to the huge number of variables given by v. Taking into account small variations for each components v_j of the vector, C function varies as in equation as presented below:

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2 + \frac{\partial C}{\partial v_3}\Delta v_3 + \cdots + \frac{\partial C}{\partial v_n}\Delta v_n$$

All diverse Δv can be incorporated in an array $\Delta v = (\Delta v_1, \Delta v_2)^T$ and all derivatives in $\Delta C = \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T$ determining the gradient of C, is possible to achieve:

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

What is very convenient in this equation is that it allows to choose Δv so as to make ΔC negative. So keeping in attention that ∇C contains how C varies for one component

v_j , is natural to designate Δv as in equation as

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

where η is again the small non negative value noted as *learning rate*. Substituting this last equation in the previous one, the successive result is obtained:

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

This equation can be seen as an update rule. It gives what is called gradient *descent algorithm*. This impressive methodology cyclically updates v attempting to minimize the cost function C, so that

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

This category of algorithm is extensively used in machine learning and needs little modification only. The main concern related to this computation is attributed to the setting of the learning rate parameter (figure 3.13): using a high value it could be possible to tackle a large number of variables and to do a quick computation without never converging to the final solution; on the other hand a too small value can lead to too much time for the training process and can make the algorithm completely unusable.

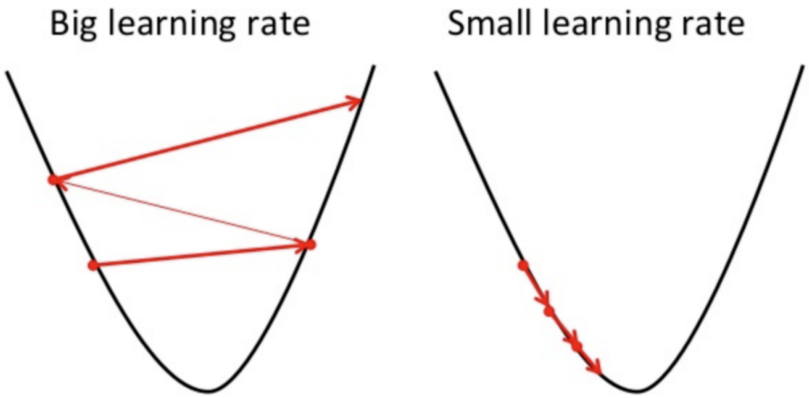


FIGURE 3.13: Representation of big and small learning rate.

The equation achieved so far are for a generic case. By adopting weights and biases in last equation instead of generic variables v, it is possible to achieve the algorithm that is actually used to train neural networks.As a matter of facts, the equation expressing C has w_j and b_j as components, such that

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

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

The cost function ($C(w,b) = \frac{1}{2n}\sum_x \|y(x) - a\|^2$), can be re-written in a compact form as $C = \frac{1}{n}\sum_x C_x$, that is simply an average over the elements $C_x = \frac{||y(x)-a||^2}{2}$ where x is an input array of the input layer. Thus, in order to obtain ∇C , the gradients ΔC_x have to be separately computed for each input x and then averaged over all the inputs, so that $\Delta C = \frac{1}{n}\sum_x \Delta C_x$

3.3.6 Stochastic Gradient Descent

This represent the first solution to the utilization of the gradient descent to a vast training input number. The *stochastic gradient descent* appreciably reduce the learning process time; rather than using all training data to figure ∇C, it is possible to evaluate its value by including a small number of ∇C_x. Instead of exploiting all the n training inputs, a subset of m of them only is used .

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

The parameter n express all training inputs and m is the picked up mini-batch of input vectors. At this point it is appropriate to rewrite the equation of weight and bias update previously shown.

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

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

The sum is no longer over all training inputs but only on the inputs of the taken mini-batch. After having updated all weights and biases one stochastically mini-batch is selected. The gradient stochastic algorithm cyclically selects a new mini-batch, in a random way, from training data until all inputs have been selected once. At this point a *training epoch* is concluded and the algorithm starts the loop again with the next epoch. The algorithm shown can have mini-batches with different dimensions and it is also possible to select a unitary mini batch. This strategy goes under the name of *online* or *incremental learning* and is analogous to how human brains function.

3.3.7 Back-propagation Algorithm

So far the most important problem in order to apply gradient descent and hence make the network learn is the calculation of the $\frac{\partial C}{\partial w^l_{jk}}$ and $\frac{\partial C}{\partial b^l_j}$. This algorithm is named after *backpropagation* since, starting from output layer of a network and getting back, it manages to compute the two partial derivatives of the cost function and afterwards compute in a straightforward way weights and biases. basically, it is a approach that allows to use gradient descent and commonly all its versions in a practical situation. Essentially, the algorithm depict a quantity known as output error δ^l_j . Adopting this notation, the output error is referred to the j^{th} neuron of the l^{th} layer. This amount

is defined with equation shown beneath. It is the version of the selected cost function with respect to the potential activation function. Once more, z refers to the j^{th}neuron of the l^{th} layer.

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

It is possible to gather all the components in a single array δ^l. So, back-propagation uses this new quantities δ_j^l, and exploits them to compute $\frac{\partial C}{\partial w_{jk}^l}$ along with $\frac{\partial C}{\partial b_j^l}$. Hence, with some mathematical passages one can extract the four central equations of back-propagation.

$$\begin{aligned}
\delta^L &= \nabla_a C \odot \sigma'\left(z^L\right) \\
\delta^l &= \left(\left(w^{l+1}\right)^T \delta^{l+1}\right) \odot \sigma'\left(z^l\right) \\
\frac{\partial C}{\partial b_j^l} &= \delta_j^l \\
\frac{\partial C}{\partial w_{jk}^l} &= a_k^{l-1}\delta_j^l
\end{aligned}$$

Where the last equation represents the selected cost function. Observing the four equations of the backpropagation algorithm, it is possible to understand how it works. Firstly, error values are computed for the output layer and then are back propagated until the input layer through the second equation. The third and fourth equations relate these previously computed errors with the partial derivatives, essential for gradient descent. It is therefore possible to use the results for the two update equations for the learning process. In conclusion, algorithms such as gradient descent or stochastic gradient descent are always related with backpropagation, that makes computations feasible. For example, using stochastic gradient descent and backpropagation, choosing a mini-batch of m training inputs, the following steps must be performed:

1. A group of m samples is chosen from the n available of the dataset;
2. For each x of the m available:
   - x is sent to the input layer;
   - *Feedforward*: following all layers $l = 02, 3, ..., L$, potentials $z^{x,l} = w^l a^{x,l-1} + b^l$ and activation functions too are calculated $a^{x,l} = \sigma(z|^{x,l})$;
   - *Output errors*: $\delta^{x,L}$ is obtained as $\delta^{x,L} = \nabla a C_x \odot \delta'(z^{x,L})$;
   - *Backpropagation*: for each layers $l = L-1, L-2, ..., 2$ are computed $\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l})$.

3. For each layer $l = L, L-1, ..., 2$ all variables are updated, along with weights and biases, according to the stochastic gradient descent algorithm;

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

4. Another mini-batch of input vectors is picked from the $n - m$ available and this is repeated until the epoch is over.

With this algorithm and using enough computational power, it is possible to perform the training of a general multi-layer network. Lamentably, this is the most basic and undeveloped algorithm and many troubles could occur, making harder the training of the network.

3.4 More on Neural Networks

Before facing next concepts, it is convenient to make preliminary remarks. The neural networks models presented previously are referred to as feed-forward neural networks, since the signal is propagated in one direction. However, there are others that are not explained in this chapter such as, long short-term memory units, recurrent neural networks, deep belief nets, Hopfield networks and others. Furthermore, it is crucial to clarify that the method used to train the introduced feed-forward neural networks is referred as *supervised learning*. Nowadays, other two main techniques are vastly used dubbed *unsupervised learning* and *reinforcement learning* respectively. However, here different problems related with neural networks training and the related possible solutions are taken into account.

3.4.1 Neuron saturation

To understand the issue we will take into account a very simplified network with one neuron only. As noticed in the previous sections, partial derivatives of the cost function $\frac{\partial C}{\partial w}, \frac{\partial C}{\partial b}$ affect how network neurons learn. Using once more the basic quadratic cost function, revised for one neuron only $C = \frac{(y-a)^2}{2}$, it is straightforward to obatain the following results:

In this simple situation is straightforward to notice that both weights and bias are driven by the activation function derivative. So, essentially, when $\sigma(z)$ is around 1 or 0, its derivative $\dot{\sigma}(z)$ assume small values. This generates a learning slowdown for the network that may prevents any improvement. The trouble can be addressed with different methods.

Learning Slowdown in the Final Layer

To solve the problem in the final layer (output layer), diverse cost functions can be exploited.

- *Cross-entropy cost function.* it is among the the most used cost functions. When multiple outputs are taken into account its expression is given by:

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

  Once again, dealing with a multi-inputs single-output single neuron unit, it can be shown that the gradient is in the the form:

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

  that suggests weights updating is directly proportional to $(\sigma(z)-y)$ and the activation function, or rather its derivative, doesn't play any role any longer.

- *Log-likelihood cost function.* It is always used along with a artificial neuron presented above: the softmax unit. As has been previously described, softmax is more and more used in the output layer in a lot of different utilizations. Indeed, the values of the output layer can be described as a probability distribution. Its cost function, known as log-likelihood cost function, follows:

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

  In this equation x is the elected training input to the network and L refers to the output a of the final layer. If the network has a good confidence about the prediction, its estimates of output value is close to one, and the corresponding cost function is assume very low values. Again, it is possible to prove that the gradients accepts the following formulation:

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

Like the method seen previously, it is possible to see that once more the two partial derivatives do not have derivative terms and hence the learning slowdown issue is avoided.

Weight and Biases Initialization

With the aforementioned solution there is an significant chance to saturate some neurons in the hidden layers. In fact, modifying the cost function only affects the output

layer. It is possible to partially overcome this issue initializing all variables with a Gaussian probability distribution having mean 0 and with a standard deviation equal to $\frac{1}{\sqrt{n_{in}}}$ where n_{in} is the number of input weights of the neuron. Using this technique, learning slowdown issue is considerably reduced, for all neuron units are less luckily to saturate. For what concerns biases initialization two main approaches can be considered: first, setting all biases equal to zero at the beginning or using the previous approach (exploiting the aforementioned Gaussian probability distribution). Certainly,barely biases influence the slowdown issue qith respect to weights and so they do not require any adjustment.

3.4.2 Data overfitting

One more important issue is represented by data overfitting. Having a large number of variables to train can lead to overfit data and rather than generalizing abstract concepts the network learns distinctivenesses of the training set. This must be averted, since a network able to perform predictions on data included in the training set only is futile. nonetheless, diverse methods can be used to overcome this critical issue.

Dividing Data

This approach include splitting available data in three different groups. *Training data* used for the learning process of the network, *Validation data* designated for testing the network at the end of each epoch and finally *Test data* used for check performances after the end of the training session. Overfitting can be at first recognized when the accuracy over the validation dataset stops increasing while the one of training data continue increasing. A straightforward strategy known as *early stopping* aim to stop training when the classification accuracy (in validation) is steady. Generally, monitoring these three class of data during training can be very convenient not only for overfitting issues likewise to set and find better *hyper-parameters* for the neural network training.

Artificially Expanding the Training Data

State-of-the-art neural networks have several parameters and so, it is quite effortless to overfit training datanot reaching the desired generalization. The easiest way to avoid this problem is to increase the number of training example. However, this option is not constantly feasible for practical reasons. Alternately, a team of researchers have developed an approach which is referred to as *artificially expanding the training data* which allows an enhancement of the dataset without the need for more training samples. The key idea is to broaden the available data set by modifying images with distortions and filters, hence operations that may reflects real world variations. Some related examples may be rotations, elastic distortions, translations, skewing.

Regularization Techniques

The approaches that have just been illustrated are one way to face over-fitting concerns, although usually they are not sufficient. Nevertheless, there is a set of methods, known as *regularization techniques*, able to reduce the aforementioned major issue also by using a fixed dataset. Next, the two most used are presented, though through the time, many papers have proposed valid alternatives. The first one is called *L1 regularization*. The practice requires a adjustment of the chosen cost function. For instance, using it with cross entropy equation presented above, lead to the addition of a parameter called *regularization* term:

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

where in the second sum w represents the number of weights and λ is a hyper-parameter that has to be chosen. It is desirable to simplify this last equation as:

$$C = C_o + \frac{\lambda}{n}\sum_w |w| \rightarrow \begin{cases} \text{if } \lambda \text{ is small the regularization term can be omitted} \\ \text{if } \lambda \text{ is large the analyzed model learns small weights} \end{cases}$$

Naturally, the proposed technique allow the model to learn small weights while they can have high values only if they notably decrease the value of chosen selected cost function. Furthermore, the regularization L1 maintain the majority of the weights values near zero and focus non-zero variables into a limited region of important connections. The second technique, which is quite popular, is commonly known as *L2 regularization*. Similar to the first one, it needs a modification of the chosen cost function, by adding an extra term.

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

Making all needed calculations is feasible to demonstrate that gradients has the following form:

$$\frac{\partial C}{\partial w}* = \frac{\partial C_o}{\partial w} + \frac{\lambda}{n}$$
$$\frac{\partial C}{\partial b} = \frac{\partial C_o}{\partial b_o}$$

Hence, gradient descent algorithms is then expressed by

$$w \rightarrow w - \eta\frac{\partial C_o}{\partial w} - \frac{\eta\lambda}{n}w = \left(1 - \frac{\eta\lambda}{n}\right)w - \eta\frac{\partial C_o}{\partial w}$$

Dropout

It is a completely different method for regularization. Dissimilarly than L1 and L2 regularization, this approach do not need any changes to be made to the picked cost function. For each training step a percentage only of neurons are activated (usually around 50%). The rest of the them is bypassed and the related weights and their biases aren't updated. At next training step the procedure is repeated using a different neurons randomly selected. Like it is explained by the same authors *"[...]this technique reduces complex co-adaptation of neurons, since a neuron cannot rely on the presence of particular other neurons"*. Exploiting this methodology, hence, a neuron is demanded-to learn more robust features that are advantageous to abstract data.

Chapter 4

Deep Reinforcement Learning

4.1 What's in this chapter

In this chapter main concepts of reinforcement learning will be explained, starting from its reasons ans sources of inspirations. Next, some required central topics will be introduced, such as MDP (Markov Decision Processes). The rest of the chapter is split in two parts: tabular methods and tabular solution methods. The former contains a very brief introduction of the three main methods used for exact solutions, especially Temporal-Difference Learning. In the latter, concerning approximate solutions methods, relevant - for this dissertation purpose - algorithms will be discussed.

4.2 Introduction

The way we learn when we're child is very similar to a trivial and error process, in which a good action is rewarded and a bad one is punished somehow; e.g. when learning how to walk, that is how to move legs in order not to fall, a wrong movement is punished with the pain of falling, while being able to stand up could lead to have better chance, more speed, a better view. This is the most natural way of learning when thinking of someone or something interacting with an environment. Any time there are no teachers, books, information, this is the way to perform a training, in any field: interaction with the environment. Interplays produce an amount of information concerning cause and effect, consequences of actions, and how to reach the goal. When this idea is taken and used in a *computational* approach, reinforcement learning is born. Hence reinforcement learning is a way to learn what to do (which actions should be taken) in order to reach better result, or rather maximise some numerical signal reward.

4.2.1 Elements of Reinforcement Learning

Beyond the environment and the agent, there are some main elements that will be almost always present in a reinforcement learning system: *policy*, *reward signal*, *value*

function and - eventually - *model* of the environment.
The *policy* of an agent defines its way of acting at any given time. Hence, it is a mapping from states detected from environment to the actions it has to take in while being in those states. Thinking of neuroscience it's easy to understand that it corresponds to stimulus-response principles or associations. The meaning is quite similar, while it representation is quite different. In Reinforcement Learning, depending on the cases, a policy could be a lookup table or a simple function, or a much more complex function involving heavy computation. Anyway, the policy is the core of reinforcement learning and it represent the most important element in a agent, since it is sufficient to define its behaviour. It must be mentioned that a policy can - and it is, in general - stochastic, i.e it may provide probabilities for actions.
A *reward signal* settle the goal in a reinforcement learning problem. At any time step, the reinforcement learning agent receives from the environment a single number, that is the signal reward. The latter hence define which events are good and which are bad, so that the agent can understand something about its actions. Indeed, its sole objective is the maximization of total amount of reward it gets over some time steps, i.e. in a whole task. Since the reward follows an action, it is a measure of how good that action has been: if an action picked out by the policy is followed by a low-value reward, then the policy may be modified in order to select others actions in the same situation, in the future.
The *value function* is much more useful to understand what is good in the long-term, i.e. in many time steps, while the reward signal gives a immediate sense of what is good. Roughly speaking, the value function indicates the total reward a reinforcement learning agent can expect to accumulate in the future, being in a certain state. Hence a state could have a high value, while having a low immediate signal reward.
However, it must be said that in general rewards are more "important", let's say primary, while values are secondary. It is easier to understand if thinking that the value of a states depends on future reward: no value would exist without rewards. Nevertheless, we are actually more interested in values when evaluating actions, since we seek actions that will lead to states with highest values, to lately get highest amount of total reward. In fact, the estimation of values is considered to be fundamentals. Likewise, methods to efficiently perform this estimations have been regarded as the most important component over last decades, for what concerns reinforcement learning.
Finally, a reinforcement learning system can also have a *model* of the environment. This of course is something that can mimics the behaviour of the environment, or rather that allows inferences over how the environment could behave. Models are especially useful for planning, i.e. the decision of a sequence of actions to be taken. When a method used to solve reinforcement learning problems has a model it is said to be *model-based*, in opposite to those without a model, that are called *model-free*.

4.2.2 Markov Decision Processes

MDPs (Markov decision processes) are a formal way to define the typical problem that a reinforcement agent has to face. A MDP is a formalization of the sequence of action that the agent has to take, considering that those action has influence not only on immediate reward, but on future consequence too, that is, on future rewards. Hence, MDPs are concerned with delayed rewards and the trade-off between delayed rewards themselves and the immediate ones. A MDP involve the presence of the elements already introduced in the previous section:

- the *agent*: it is the learner and the decision maker;
- the *environment*: it comprises anything outside the agent. The latter interacts with the environment itself.

The agent selects and action and the environment gives back a new situation for the agent and a reward, which the agent has to maximize. This is repeated continually. A scheme can be seen in figure 4.1.

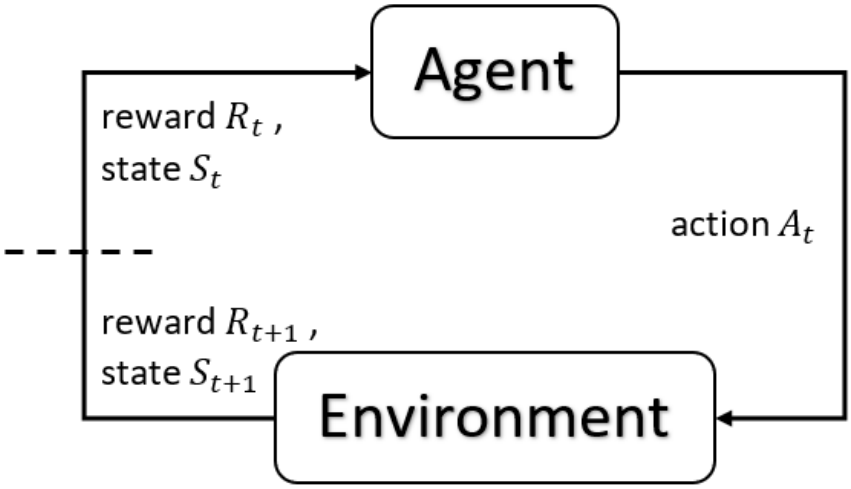


FIGURE 4.1: Scheme of a Markov Decision Process.

These two elements keep interacting and this exchange of input and outputs goes on. We consider them interacting at discrete time steps t , with $t = 0, 1, 2, 3, ...$ At each time step the agent receives some information about the environment's state $S_t \in \mathcal{S}$, then it's select and action $A_t \in \mathcal{A}$ depending on that state. At the next time is in a new state, S_{t+1} and it receives a *reward* $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ as a numerical value. Hence an agent and a MDP give birth to a sequence such as the one following:

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

where parenthesis are added just to separate environment and agent outputs. A scheme to understand the sequence is given below (figure 4.2).

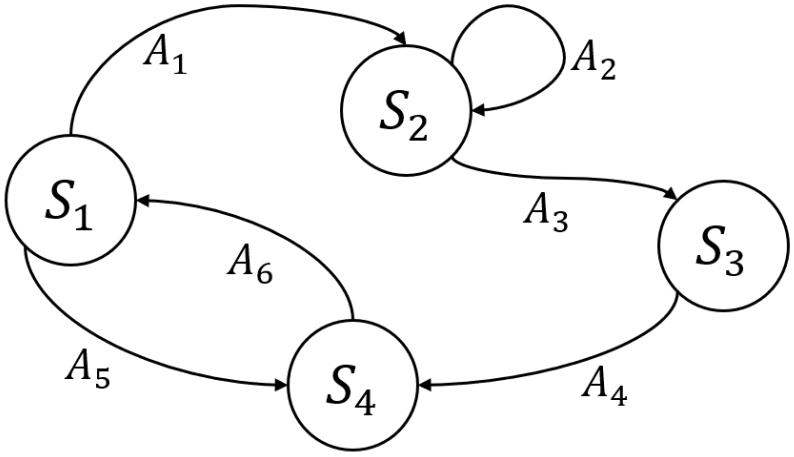


FIGURE 4.2: Example of scheme of a finite Markov Decision Process.

An MDP may be *finite* when the numbers of states, actions and reward are finite number ($\mathcal{S}, \mathcal{A}, \mathcal{R}$ have a finite number of elements). In a finite MDP it exist a clearly defined discrete probability distribution for the random variables R_t and S_t depending only on preceding state and action. It means that a for particular values of these variables ($s' \in \mathcal{S}$ and $r \in (R)$, we have a probability for them to occur given the values of preceding state and the chosen action. Such distribution is p:

$$p(s', r|s, a) \doteq Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\},$$

This definition is important because it characterize MDPs' environments' dynamics. If (i) the probabilities of any value of S_t and R_t depends exclusively on state and action (S_{t-1} and A_{t-1}) that come immediately before it's known and (ii) the state includes - completely - information about the aspects of the previous interaction between agent and environment that make a difference in future steps, the state has the *Markov property*. Thoght it has been assoumed the MDP to be *finite* this property is considered to be true also when it should not, for instance in next section, where approximate solution methods for (*non-finite*) MDP will be taken into account.

Rewards and Goals

As has already been introduced, in reinforcement learning, the goal or purpose of the agent is expressed in terms of a signal, that has been called *reward*. Since the agent gets a reward at each time step, it's aim is to finish is task or life with the highest amount. That can be stated as the "*reward hypothesis*" [12] :

> "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 idea of signal reward is a distinctive feature of reinforcement learning. This formulation of goals could appear limiting, however it is not. For instance, image an agent that has to learn how to escape from a labyrinth. One way to provide reward could simply be to give -1 at each time step but when it gets out. In this way it would be motivated to escape as soon as possible but would no give a clue about how to do that. There are a lot of possible rewards one might think of.

Agent Return

It has already been said that the agent's goal, in reinforcement learning, is to maximize the reward. This final value can be called *return* and can be denoted as G_t. However the definition of G_t is not unique and it deepens on the type of problem.

In the simplest case it can be defined as follows:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T,$$

Note that it has a terminal reward. Indeed, this approach can make sense when the agent-environment interaction is divided into subsequences (the *episodes*) and it has a final step notion. Some examples could be episodic games. These are referred as *episodic tasks*

Another case is represent by agent-environment interactions that cannot be broken into episodes. In this type of task there could be no end. Examples could be control of always active processes. These are called *cotinuing tasks*. In this case there would be $T = \inf$. Hence the return could diverge to infinite and lost sense. The concept of *discounting is introduced* is to overcome this issue. The *expected return* is then defined as follows:

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

where the *discount factor*, or *discount rate*, γ is a parameter such that $0 \leq \gamma \leq 1$. Thanks to this parameter the return, after a infinite number of time step, has a non infinite sum and the k-th reward is only woth γ^{k-1} times with respect to the current reward. If $\gamma = 1$ the situation is the same of above, while if $\gamma = 0$ the agent is said to be "myopic" since it only "sees" the immediate reward and hence that's the only reward it tries to maximize.

Value Function, Policies and Optimality

Though the meaning of *value functions* has already been introduced, now a deeper definition is given. A state value function estimates how good is for the agent to be in a certain state, and this is done in terms of expected return. Also a state-action

pair value function can be define, that estimates how good is to choose a certain action in a given state. Also in this case the "how good" is related to the expected return. A *policy* is a function that maps from to states to the probabilities of choosing each possible action. Hence, if the agent follows a policy π, $\pi(a|s)$ is the probability that if $S_t = s$ then $A_t = a$. Given a state s, its *value function* under a policy π, $v_\pi(s)$, is the expected return when the agent starts in s and follows the policy π from there on. In MDPs:

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

where $\mathbb{E}_\pi[\cdot]$ is the expected value operator given that the agent follows the policy π. v_π is *the state-value function for policy* π.
Quite similarly it can be defined the *action-value function for policy* π, q_π:

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

It is straightforward to define optimal policies and optimal value functions. A policy π is better than another π' if the expected return of the former is greater then the latter's, hence:

$$\pi > \pi' \iff v_\pi(s) > v_{\pi'}(s)$$

Of course if there's a policy that is better than (or equal to) all other policies, it is optimal and it's denoted as π_*. There could be more optimal policy, however the notation is the same. The all would share the *optimal state-value function*, v_*:

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

Also *optimal state-action value function* is shared:

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

Optimality cannot be achieved in non-finite MDPs due to many practical reasons (e.g. memory required) . Optimal policies and value functions can be approximated.

4.3 Tabular methods - Reinforcement learning

It is convenient to first face reinforcement learning in simplest cases, i.e. in tabular cases, or when finite MDPs are involved. In this section the three methods, or rather collections of methods, used for solving these kind of reinforcement learning problems will be shortly discussed. At first Dynamic programming will be taken into account, only for completeness, since it is the less applicable and has no role in this dissertation. Then we will briefly consider Monte Carlo methods. Finally Temporal-Difference

Learning will be introduced, which can be said to be the central idea of reinforcement learning.

4.3.1 Dynamic programming

Dynamic programming (DP) can be referred to a set of algorithms that can be used for computing optimal policies when a perfect model of the environment as an MDP is given. It's easy to understand that DP may not be much useful in most cases, since a perfect model does not exist. Often, even it would exist, the computational expense would be too high for DP to be used. A field where DP may be used is the financial one. Usually DP include some methods that are often combined.

Iterative policy evaluation is a method for computing the state-value function v_π given an arbitrary policy π. This is done by using the following equation:

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

that is known as **Bellman equation** for v_π. This equation can be derived from those seen in the previous section. In DP, the Bellman equation is used as an update rule. Given an initial approximation v_0, successive ones can be obtained:

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

No further details are given concerning iterative policy evaluation. After the value function v_π has been determined, it is convenient to find an optimal policy, since an arbitrary one has been used for finding v_π. It it done exploiting already seen relations. Shortly, a new*greedy* policy, denoted by π', is given by

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

To this follows another iterative process where both π and v_π are optimized. A better treatment of DP can be found in [12], where also others algorithms are discussed.

4.3.2 Monte Carlo methods

Monte Carlo (MC) methods in contrast with DP require just *experience*, that is, arrays with states, actions, rewards. These may come from interaction with a real or a simulated environment. Learning with a real environment has the huge advantage that no prior knowledge of the environment's dynamic is needed. However, most of the time it may be convenient to use a simulated environment. Anyway also in this case it not

required a complete knowledge of the environment's dynamic. Indeed, it is required a model but not the probability distributions related to all possible transitions, which are required if one wants to use DP.
Monte Carlo methods attempt to solve reinforcement learning problem by using sampling and averaging of state-actions pairs returns. In the following episodic tasks are considered, to have well-defined returns.
As done for DP, at first the prediction problem is considered, that is, the computation of state value v_π and state-action value q_π given an arbitrary policy π. The idea behind Monte Carlo methods is to average all the return that has been seen in different visits in a certain state. As the number of these visits increases, the average value should converge to the expected one. There are two main methods for prediction:

- first-visit MC method makes the estimation of v_π by averaging all the returns that follow the first visits, i.e. if in a episode a state is visited twice the return related to this second visit is not considered.
- every-visit MC method average all the returns, also those ignored by first-visit MC method.

The pseudo-code of first-visit MC method is given below. The other is identical but for the condition of state at line 6.

Algorithm 1: First-visit MC for estimating $V \approx v_\pi$

Input: a given fixed policy π, a positive integer n_e (number of episodes)
Output: the value function V

1 Initialization of $N(s) = 0$, $\forall s \in \mathcal{S}$ `// N is a counter`
2 Initialization of $returns(s) = 0$, $\forall s \in \mathcal{S}$
3 **for** *episode* $e \leftarrow 1$ ***to*** $e \leftarrow n_e$ **do**
4 Generate, following π a sequence $S_0, A_0, R_1, S_1, A_1, R_2, ..., S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$ **for** *timestep* $\leftarrow T - 1$ ***to*** $t \leftarrow 0$ **do**
5 $G \leftarrow G + R_{t+1}$
6 **if** *state* S_t *is* ***not*** *present in the sequence* $S_0, ..., S_{t-1}$ **then**
7 $returns(S_t) \leftarrow returns(S_t) + G_t$
8 $N(S_t) \leftarrow N(S_t) + 1$
9 $V(s) \leftarrow \frac{returns(s)}{N(s)}$, $\forall s \in \mathcal{S}$

When a model is not known, it is more useful to estimate the state-action pair $q(s, a)$ and it is done in the same way. A state-action pair (s, a) is visited when the agent visit the state s and then takes the action a. The difference now is that if a policy is deterministic some actions could never be taken; hence, to avoid that any pair is never visited, a non-zero probability is given at the start to each of them. This guarantees that in an infinite number of time steps any pair is visited. This assumption is called

exploring starts. However this cannot always be done and the best practice is not to have a deterministic but rather a stochastic policy, with a non-zero probability to choose any action in any state.
Concerning Monte Carlo *control*, that is the approximation of optimal policies, it is followed the idea of *generalized policy iteration* (GPI) which is actually followed in DP too. IN GPI they are maintained both an approximate value function and an approximate policy. They are altered repeatedly: q is altered to approximate more closely the value function for the current policy and π is repeatedly improved depending on the current value function. How a policy is improved? It is done by exploiting the *policy improvement theorem*, that states that if

$$q_\pi(s, \pi'(s)) \geq v_\pi(s)$$

then policy π' is as good as, or better than, policy π. It is the same to say the it must a greater or an equal return from all states $s \in \mathcal{S}$

$$v_{\pi'}(s) \geq v_\pi(s)$$

To understand these relations, one could imagine that a some time-steps instead of following the policy π, another one,π' , is followed, hence another action is chosen. This could bring to the discovery of a better policy if in the end the return is greater. Anyway, as seen in section dedicated to DP, a new greedy policy, can be found:

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

The improvement is repeated as shown in figure 4.3

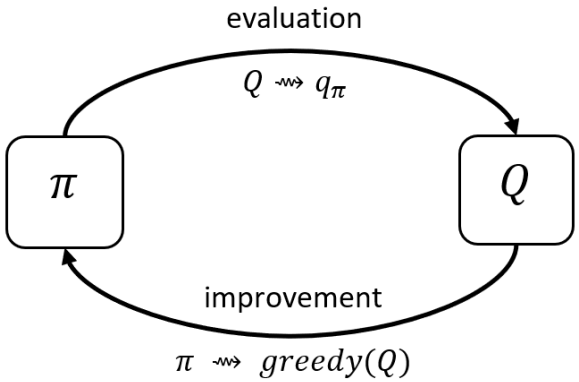


FIGURE 4.3: Scheme of policy improvement.

However, this procedure makes sense only considering *exploring starts* and a huge number of episode, which may be infeasible. A possible solution is to use ε-greedy policies, such that

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

It is useful to define **on-policy** methods and **off-policy** methods, also for the rest of the chapter. On-policy methods are those that try to improve or evaluate the policy that is used to make decision, while off-policy methods do the same but on a policy that is different from the one used to generate the data.
So far on-policy methods have been taken into account. Concerning off-policy methods, suppose we want to estimate v_π or q_π from sequences $(S_0, A_0, R_1, ...)$ generated by a policy b instead of our policy π. These two may be called *behaviour policy* and *target policy*, since the former is the one actually behaving (i.e. acting during the episodes, is also more exploratory) and the latter that should be learned and become optimal. Off policies usually uses *importance sampling*, that is a way to weight the returns according to their occurrence probability. In MC methods this is required because averaging returns that come from policy b would not lead to a correct result, that is v_b instead of v_π. More details concerning off-policies and importance sampling will be given later.

4.3.3 Temporal-Difference Learning

Temporal-difference (TD) learning is the central idea of reinforcement learning and can be seen as a combination of DP and MC ideas. As DP methods, TD's can update estimates without waiting for a final outcome (these methods bootstrap), by using other learned estimates. Moreover, as MC methods, TD's learn from raw experiences and do no need a complete knowledge of the environment's dynamics.
TD methods can update after just one time step, hence if a MC method is:

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

with G_t return after time t and α a parameter that gives a weight to the update. A TD method instead could be as follows:

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

where α has the same meaning. Hence for TD the target is $R_{t+1} + \gamma V(S_{t+1})$. This method is called TD(0), also known as *one-step* TD. The TD *error* is defined as follows:

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

TD methods have some straightforward advantages with respect both to DP and MC methods. The no need for a model has already been cited, bootstrapping is a huge advantage, especially when the episodes are very long or have no end at all. Furthermore, TD method are less susceptible to some MC-related problems such as the need to ignore or discount those episodes in which experimental action are taken. So TD methods are often convenient, also because they are safe, since they have proved to converge in good situations and in general faster than MC ones. Convergence proofs

are not treated in this dissertation. Next main TD methods will be introduced, mainly as pseudo-codes.

SARSA

The SARSA algorithm is an on-policy TD method that takes its name from the sequence that has already been seen many times:$(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$. This sequence is called a quintuple. SARSA aims to keep estimating q_π for the behaviour policy π, and as already seen, to push π toward being greedy to respect of q_π. The pseudo-code of SARSA is shown below:

Algorithm 2: SARSA

Input: small $\varepsilon > 0$ for policy π (e.g. ε-greedy to Q), step size $\alpha \in (0, 1]$

Output: the state-action value function Q

Initialization of $Q(s, a)$, $\forall s \in \mathcal{S}^+, a \in \mathcal{A}(s)$ with $Q(terminal\,state, \cdot) = 0$
for *each episode* **do**
    Initialization of S
    for *each step in the episode* **do**
        Selection of A using policy
        Perform A and observe R, S'
        $Q(S, A) \leftarrow Q(S, A) + \alpha\,[R + \gamma Q(S', A') - Q(S, A)]$
        $S \leftarrow S'$
        $A \leftarrow A'$
        if S *is terminal* **then**
            break while

Q-Learning

Q-Learning algorithm is defined by :

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

hence the value function Q directly approximate the optimal action-value function q_*, with no dependence on the policy being followed.

Algorithm 3: Q-Learning

Input: small $\varepsilon > 0$ for policy π (e.g. ε-greedy to Q), step size $\alpha \in (0, 1]$

Output: the state-action value function Q

```
Initialization of Q(s, a), ∀s ∈ S⁺, a ∈ A(s) with Q(terminal state, ·) = 0
for each episode do
    Initialization of S
    for each step in the episode do
        Selection of A using policy
        Perform A and observe R, S'
        Q(S, A) ← Q(S, A) + α [R + γ max_a Q(S', a) − Q(S, A)]
        S ← S'
        if S is terminal then
            break while
```

Further details on Q-Leaning will be given later.

4.4 Approximate Solution Methods - Deep RL

The previous section concerned tabular methods, for simplicity. In this part of the chapter those methods are extended to problems with larger state spaces. Indeed, in many of the possible task that could be faced by reinforcement learning the state space can be really enormous. As already announced, in such cases it cannot be actually expected to find an optimal policy, nor an optimal value function, even considering a infinite amount of time and data. Hence there's the need for good approximate solution in accordance to available computational resources. Here comes the concept of generalization, that is, to have a good approximation for a huge subset, produced thanks to the experience of a much smaller subset. This is done with *function approximation*. There are different methods that can be used for function approximation, for instance tile coding, kernel-based. However in here only ANN (artificial neural networks) will be taken into account.

In the previous section, the value function was represented by a table; here this is no longer possible. It is now represented by as parametrized functional form having a weight vector $\mathbf{w}$ ($\mathbf{w} \in \mathbb{R}^d$). For ANN $\mathbf{w}$ is represents the parameters of the net, or rather, its weights. The state value function is now written as $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$, since it is an approximate value of state s depending on vector $\mathbf{w}$. In the same way, one could write $\hat{q}(s, a, \mathbf{w}) \approx q_*(s, a)$.

In the following first some elements or concepts are introduced, then a brief description of these algorithms is present.

4.4.1 Experience Replay

Recently *experience replay* has been introduced in the world of RL. It has been used at first in the DQN algorithm [13], that will be described later. Experience replay involves storing at each time-step the experiences made by agent. They are saved as a tuple $e_t = (s_t, a_t, r_t, s_{t+1})$ in a data-set or array $\mathcal{D} = e_1, e_2, ..., e_N$, that is called *replay memory*. At each-time step the agent performs an action following its policy and then the network is trained using a fixed length input with the histories - or rather with a representation of them - taken from the replay memory. The use of this technique lead to many advantages, such as the data efficiency given by the possibility of re-use the same experience for many weight updates. Moreover, since the choice of the experience to take from the replay memory is stochastic, the learning process itself is more efficient, because strong time correlations are avoided (in standard on-line algorithms the weight update is done using consecutive samples). The third advantage is that in on-policy algorithms current parameters determine the following samples that are then used to train the parameters themselves. Also this loop is avoided with experience replay. It should be clear that the use of this technique lead to an off-policy learning.

4.4.2 Target Network

Another technique widely used involve the use of a *target network* that is used a reference. This is done to increase convergence since in this way the on-line network - no matter what's approximating - has a fixed reference to follow when using the TD-error. The target network is updated after a certain amount of steps. This concept has already been introduced when taking into account Monte Carlo Method, at the beginning of the section.

4.4.3 Actor-Critic Architecture

The Actor-Critic architecture involve the use of two different entities, e.g. two function approximators, that compose the agent as we know it from the formal definition of an MDP. The actor is the part that chooses the action to be performed, while the critic judges what the actor does and tells it how good it's what it is doing. When talking of Actor-Critic architecture, is convenient to remember that it is used only during the training. When it is over, only the actor is used. As a matter of fact the critic is used to the train the actor, even though the critic requires a training too. A scheme of the architecture is provided in figure 4.4.
The critic may approximate the action-value function using the TD error and the actor may approximate the policy. While the critic usually uses TD error loss function gradient to update its parameters, the actor network may be updated exploiting gradient coming from critic, e.g. using Policy Gradient algorithm. That is why the critic is said to "teach" the actor how to behave. A policy gradient algorithm is discussed later.

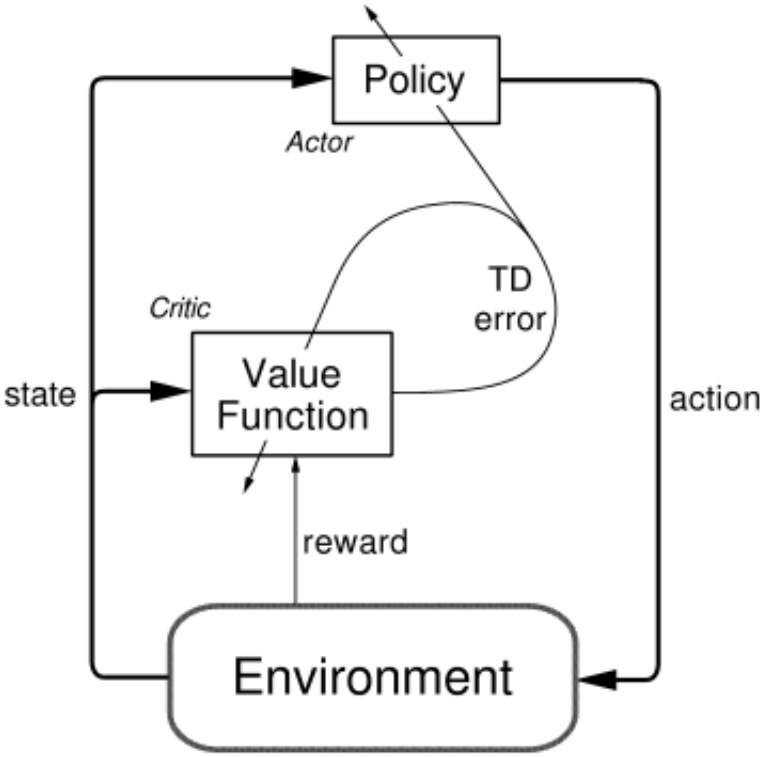


FIGURE 4.4: Scheme of Actor-Critic architecture.

4.4.4 DQN Algorithm

This and many other algorithms were at first presented as games solver, for demonstration purposes. This is also due to the chance to see them as finite MDP, that have already been introduced. The DQN (Deep Q-Learning Network) is an evolution of standard Q-Learning methods. Hence it exploits the definition of the optimal-value function,

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

and the Bellman equation for the action-state value:

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

It is actually infeasible to exactly approximate it, so - in DQN - neural networks are used as function approximator, $Q(s,a;\theta) \approx Q^*(s,a)$, where θ are the weights of neural network. The Q-network is trained performing the minimization of a loss function $L_i(\theta_i)$, that is different at each time-step,

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

where y_i is no more than the target for the current iteration i, while $\rho(s,a)$ is a probability distribution. The loss function is usually optimized by stochastic gradient descent.

This algorithm is model-free, off-policy and does not provide a policy, but only an approximation of the state action value function. The policy is an ϵ-greedy one with $\epsilon - 1$ chance of choosing a random action in the set. The pseudo-code is provided below.

Algorithm 4: Deep Q-Learning with experience replay

Input: small $\varepsilon > 0$ for policy π (ε-greedy), replay memory capacity N, number of episode M

Output: Function approximator for state-action value function Q

Initialization of replay memory $\mathcal{D}$ with capacity N, initialization of Q function approximator with random weights
for *episode in* M **do**
  for *iteration in* T **do**
    Generate random number $0 \leq h \leq 1$
    if $h \leq \epsilon$ **then**
      Select a random action a_t
    else
      Select $a_t = max_a Q^*(s_t, a; \theta)$
    Execute action in the (simulated) environment and observe r_t, s_{t+1}
    Set $s_t + 1 = s_t$
    Store experience transition (s_t, a_t, r_t, s_{t+1}) in memory $\mathcal{D}$
    Take a random mini-batch of transitions (s_j, a_j, r_j, s_{j+1}) from replay memory $\mathcal{D}$
    Set target: $y_j = \begin{cases} 1 : r_j, & if final state \\ 0 : r_j + \gamma max_{a'} Q(s_{j+1}, a'; \theta) \end{cases}$
    Use a gradient descent step on loss $(y_i - Q(s, a, \theta_i))^2)$

4.4.5 DDPG Algorithm

There are different version of deterministic policy gradient (DPG), involving many features, architectures, type of learning. In the following it is considered to have an Actor-Critic architecture and to be off-policy. Policy gradient methods use gradient ascent on the policy performance objective function J,

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

that is, the cumulative discounted reward. When referring to off-policy DPG, it can be written [14]:

$$\begin{aligned} J_\beta(\mu_\theta) &= \int_{\mathcal{S}} \rho^\beta(s) V^\mu(s) ds \\ &= \int_{\mathcal{S}} \rho^\beta(s) Q^\mu(s, \mu_\theta(s)) ds \end{aligned} \tag{4.1}$$

where β is the behaviour policy ($\beta(a|s) \neq \mu_\theta(a|s)$) that is used to sample the transition, μ_θ is a deterministic policy with parameters θ. Its gradient - approximately - is the following,

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

True action-value function is replaced with a differentiable counterpart $E^w \approx Q^\mu$. The basic steps for DPG are then:

$$\begin{aligned} \delta_t &= r_t + \gamma Q^w(s_{t+1}, \mu_\theta(s_{t+1}) - Q^w(s_t, a^t) \\ w_{t+1} &= w_t + \alpha_w \delta_t \Delta_w Q^w(s_t, a_t) \\ \theta_{t+1} &= \theta_t + \alpha_\theta \Delta_\theta \mu_\theta(s_t) \Delta_a Q^w(s_t, a^t)|_{a=\mu_\theta(s)} \end{aligned}$$

Considering the use of neural networks as function approximator, hence now referring to the DDPG algorithm [15] the loss function can be formulated as done for Q-Learning,

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

where y_t is the target,

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

The concept of target network is used in this algorithm to increase convergence properties. Hence there are two copies both of actor and critic used for the optimizing process. They are denoted as

$$\begin{aligned} &Q'(s, a|\theta^{Q'} \\ &\mu'(s|\theta^{\mu'}) \end{aligned}$$

They are updated after a certain amount of time steps. While DDPG involve a soft update, during the implementation an hard one is performed.

$$\begin{aligned} \theta^{Q'} &\leftarrow \theta^Q \\ \theta^{\mu'} &\leftarrow \theta^\mu \end{aligned}$$

Another huge difference with respect to literature DDPG is related to exploration/-exploitation dilemma. To overcome the problem there's always a non-zero probability to perform a random action. In the original DDPG a noise $\mathcal{N}$ is used instead. The pseudocode of the DDPG version used in this thesis is provided below.

Algorithm 5: DDPG algorithm

Input: small $\varepsilon > 0$ for exploration, replay memory capacity N, number of episode M, $update_step$ for target networks update

Output: Function approximator for state-action value function Q

Random initialization of critic and the related target network, respectively $Q(s, a|\theta^Q)$ and$Q'$with$\theta^{Q'} \leftarrow \theta^Q$

Random initialization of actor and the related target network, respectively $\mu(s, a|\theta^\mu)$ and$Q'$with$\theta^{\mu'} \leftarrow \theta^\mu$

Initialization of replay memory $\mathcal{D}$

for *episode in* M **do**

  Receive initial state s_1 **for** *iteration* t *in* T **do**

    Generate random number $0 \leq h \leq 1$

    if $h \leq \epsilon$ **then**

      Select a random action a_t

    else

      Select $a_t = \mu(s_t, \theta^\mu$;

    Execute action in the (simulated) environment and observe r_t, s_{t+1}

    Set $s_t + 1 = s_t$

    Store experience transition (s_t, a_t, r_t, s_{t+1}) in memory $\mathcal{D}$

    Sample a random mini-batch of transitions (s_j, a_j, r_j, s_{j+1}) from replay memory $\mathcal{D}$

    Set target: $y_j = \begin{cases} 1 : r_j \;\; if final state \\ 0 : r_j + \gamma Q'(s_{j+1}, \mu'(s_{j+1}|\theta^{\mu'})|\theta^{Q'} \end{cases}$

    Update critic minimizing loss: $L = (y_i - Q(s_j, a_j, \theta^Q))^2)$

    Update actor's policy using the policy gradient:

$$\Delta_{\theta\mu} J \approx \Delta_a Q(s, a, |\theta^Q)|_{s=s_j, a=\mu(s_j)} \Delta_{\theta\mu} \mu(s|\theta^\mu)|_{s_j}$$

    if $global step \,\%\, update step == 0$ **then**

      $\theta^{Q'} \leftarrow \theta^Q$

      $\theta^{\mu'} \leftarrow \theta^\mu$

Chapter 5

Ultra-Wideband

5.1 Localization techniques

When moving in a certain environment, both during exploration or a precise task, localization is a key concept. Also human need to know their position with respect to something; it may be not obvious, since we're used to it. It's easier to understand this concept if thinking of walking indoor with the eyes closed. Anyone would try to localize itself into the house by using the other senses. Localization can be done thanks to positioning systems.

5.1.1 Classification

Positioning systems can be classified, based on reference frame, type of measurements, network configuration.

Classification based on reference frame

This classification is based on the the reference frame to which the positioning system refers. There are two categories:

- *Global positioning*: such systems allow a positioning all around the globe (the reference frame is absolute).
- *Local positioning*: the user is allowed to know its position with respect to some local landmark or infrastructure (the reference frame is relative).

Classification based on measurement type

Positioning system perform localization thanks to a process that is usually made of two main steps. First a measurement is taken, then, thanks to such information, the position can be computed, using different algorithms. In the following, main techniques of ranging (that is, the aforementioned measurement) are shown:

- *Angle-of-Arrival (AOA)* : in this case, the knowledge of the angle of arrival of the signal is exploited, hence the direction.
- *Received signal strength (RSS)* : such techniques are range based and can acquire information by measuring the power of the arriving signal.
- *Time of arrival (TOA)*: they are range difference based. The propagation delay of the signal is measured.
- *Near-field ranging (NFR)*: this range based technique combine knowledge of electric and magnetic field for relating distance to the angle between them in near-field conditions.

Classification based on network configuration

Another classification can be obtained by considering the disposition of the nodes (basically the devices that allows the measurements to be taken, such as antennas), or rather on the network configuration.

- *Anchor-based*: the location of some nodes must be known.
- *Single-hop*: The distance with respect to the anchors is obtained by direct interaction.
- *Multi-hop*: The distance with respect to the anchors is obtained thanks to intermediate nodes.
- *Anchor-free*: Nodes position is unknown, while relative coordinates can obtained.
- *Range-free*: connectivity-related information only is exploited.
- *Network-centered*: Nodes can compute their position only after receiving information from the target.
- *Terminal-centered*: Nodes can computed their own position after acquiring the distance measurements.

5.2 Ultra-wideband

5.2.1 Introduction

Ultra-wideband (UWB) is indoor positioning which has recently shown to be quite promising. It allows accurate distance estimation in short distanced. Moreover it has some attractive features, such as being low power consumption, low cost, high-rate transmission and low complexity related. In UWB very short duration impulses are used, for which the energy is spread quite uniformly on a wide frequency band ranging

that goes from very low frequencies to gigahertz. This method, impulse radio UWB (IR-UWB) has the advantage of allowing a really precise estimate of the TOA, hence of the positioning.

5.2.2 Technology basic theory

UWB is allowed in the frequency range of $3.1 - 10.6\ GHz$, moreover the definition of UWB signal has restrictions on bandwidth and frequency or on fractional bandwidth (it should exceed 20%) [16]. A UWB system is made up by three basic components:

- *Transmitter*: the element generating the electromagnetic waves, producing the radio signal.
- *wireless channel*: it is the means by which the signal propagates.
- *Receiver*: it is the device receiving the transmitted signal.

No details are given concerning the channel model, the impulse signal and the devices architecture, since it is out of this dissertation interest. However, time-based ranging concept are given. UWB exploit the time of arrival (TOA) for estimating the distance. In line-of-sight (LOS) conditions (hence with no obstacles between transmitter and receiver) the distance d can be computed by measuring the path delay τ_0 as

$$d = c \cdot \tau_0$$

with c speed of light. However this can easily done just considering first path. In general a UWb systems receives a signal that can be expressed as

$$r(t) = \sum_{n=1}^{N} a_n s(t - \tau_n) + n(t)$$

where s() is transmitted signal, n(t) is the white noise. UWB systems,having a multipath resolution, allows an estimation of first path delay τ_0 and also of the other τ_n parameters. Furthermore, UWB exploit the so called Two-way-ranging (TWR) protocol for computing the time of flight. The two nodes, transmitting and receiving the signals, reset their clocks when receiving the signals (as shown in figure 5.1)
The time of flight is computed as:

$$t_prop = \frac{1}{2}(t_round - T - t_{off,1} - t_{off,2})$$

in which t_round is the total time between node 1 transmission and reception. $t_{off,1/2}$ is the time errors related to detecting of the signals.
There are some sources of error in UWB positioning, such as multipath propagation, high time resolution, multiple access interference and NLOS propagation. The latter

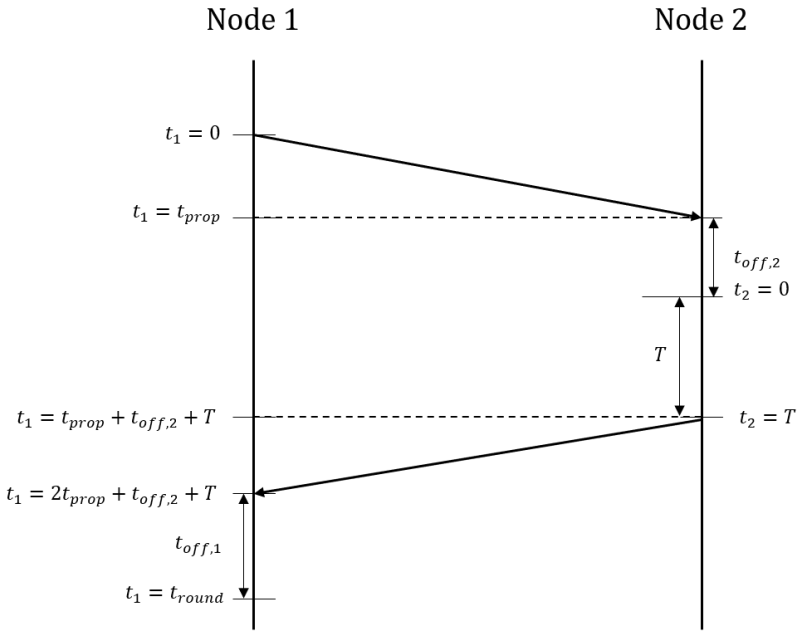


FIGURE 5.1: Two-way-ranging scheme.

occurs when there's an obstacle in between transmitter and receiver. This lead to positive bias related to the reflected paths. Basically NLOS measurements should be discarded, hence distinguishing those from LOS ones.

5.2.3 Position estimation

The aforementioned system is able to measure the distances between the target node and the anchors (or anchor nodes). by using this information the position can be computed. Since the concept is quite similar, in the following a 2D case is illustrated. The target position is computed as intersection of the circle around the anchors, having radius which values are equal to the previously measured distance d_i. The anchors have known coordinates (x_i, y_i), which are also the centres of the aforementioned circles. A representation of the problem is given in figure 5.2.
The actual problem to be tackled is in 3D. However the main difference is that the circles are spheres and their minimum number is 4. The coordinates to be found are then (x, y, z) and their values are obtained as solution of the system of equation:

$$\begin{cases} (x-x_1)^2 + (y-y_1)^2 + (z-z_1)^2 = d_1 \\ (x-x_2)^2 + (y-y_2)^2 + (z-z_2)^2 = d_2 \\ (x-x_3)^2 + (y-y_3)^2 + (z-z_3)^2 = d_3 \\ (x-x_4)^2 + (y-y_4)^2 + (z-z_4)^2 = d_4 \end{cases}$$

This equation can be rewritten as

$$\mathbf{Ax} = \mathbf{b}$$

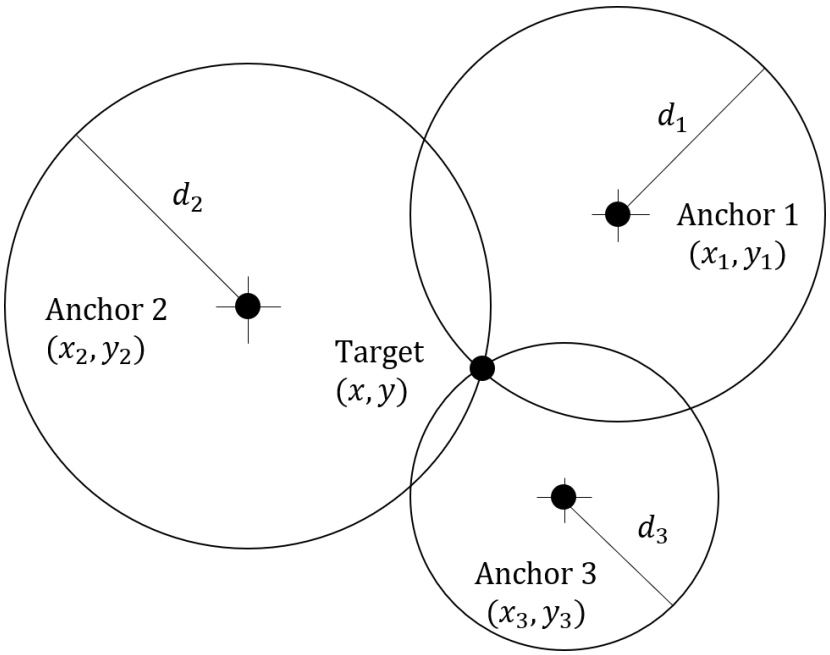


FIGURE 5.2: Triangulation scheme.

where $\mathbf{x} = [x, y, z]$ and it can be solved using linear least squares (LLS) method. However when used in this work, another algorithm is used, that is Gauss-Newton method. The solution is given by an iterative process aimed to solve non-linear least squares approximation problem. The solution is the one minimizing the sum of all squares of residuals:

$$S = \sum_{i=1}^{m} r_i^2$$

the *i-th* residual is

$$r_i = d_i - \sqrt{(x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2}$$

At each iteration, a new value of the coordinates is returned by the algorithm:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - (\mathbf{J}^T\mathbf{J})^{-}1\mathbf{J}^T\mathbf{r}(\mathbf{x}_k)$$

where $\mathbf{r}$ is a vector function depending on residuals $\mathbf{J}$ is the related Jacobian matrix. This method keep iterating until the error is under a certain tolerance on based on a maximum number of steps.

Chapter 6

Platform and implementation

6.1 What's in this chapter

This chapter is basically dedicated to the applicative part of the work, that is, to hardware and software, which they are and how they are used. In the first segment, a description of hardware components is given; in other word the *material* part of the system is taken into account. Next, software is considered, that is, all the programming tools, languages, programs *et cetera*. Finally there's a section devoted to making clear how software and hardware are merged and how the overall system works.

6.2 Robot platform

The robot platform is the set of material components used to implement the software. For this work there are two main elements. First there's the mobile robot, that is the set of mechanical structure, actuators, battery, sensors etcetera. Then Ultra-wideband components are taken into account separately.

6.2.1 TurtleBot3

For this work a developing platform has been used, that is called TurtleBot3. It includes different mobile robots that are in general easily customizable, modular and compact [17] [18]. In figure 6.1 a scheme of Burger, the model that has been used, is provided. TurtleBot3 is design to work with ROS (Robot Operative System) which is basically a sets of frameworks for developing and programming of robots. For instance, it takes car of communicating information and data. Further information on ROS are given in next section.

Sensors

Sensors are devices that can somehow measure something, or rather that give a measurable response to some change in physical quantities. In the robotics context they

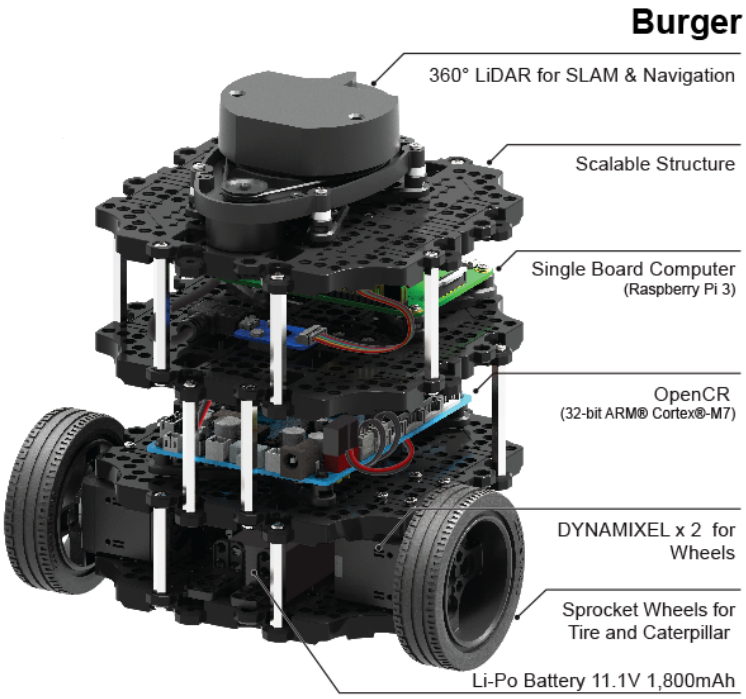


FIGURE 6.1: Scheme of the TurtleBot3 model "burger" and its components [19].

can be used to measure quantities related to the internal state of the robot (these are said to be *proprioceptive*) or concerning external state of the robot (and they are said to be *exteroceptive*) The Burger is equipped with different sensors: IMU sensors (gyroscope, accelerometer, magnetometer), Lidar, motors encoders.

IMU (Inertial measurement unit) includes a 3 axis **gyroscope** which is able to measure the rotational speeds along the three axis x, y, z.

A 3 axis **accelerometer** is also provided by IMU. It is able to measure the acceleration to which the sensor is subjected , along the same axis as the gyroscope.

Finally the IMU has also a 3 axis **IMU magnetometer** that gives the value of the magnetic field along the axis x,y,z. It basically is measure the orientation of the sensors, hence of the robot, with respect to the magnetic pole.

The burger is also equipped with a **LDS** (**Laser Distance Sensor**). It is used through a 360 degrees **LiDaR** (Light Detection and Ranging), that is a technique to measure distances all around on a plane (when a 2D lidar is taken into account). Basically the measure is performed by illuminating an object and capturing with a sensor the reflected light. The distance is computed using differences in time laser returns and wavelengths. With lidar, this is repeated along 359 directions equally - theoretically - spaced (hence from 0 to 359 degrees) by rotating the LDS. The latter, for the Burger,

is rotated a $5Hz$.

The burger has two motors with precise **encoders**. The encoder are used to measure motors or wheels rotation (they are mechanically coupled). This information is then used, via a model of the robot, to estimate its position in time. The encoders can exploit different technologies, e.g. they can be magnetic (as Burger's ones) or optical.

Dynamixels

Concerning the movement of the robot, it is made possible thanks to dynamixels, that are small actuation systems including a DC motor, a reduction system and integrated controller, drivers and network. They are high quality and performances smart actuators that are widely used in robotics (see figure 6.2) [20].

FIGURE 6.2: Gamma of available dynamixels.

Open CR

OpenCR (Open-source Control Module) is an embedded board used in all TurtleBot3 models that takes care of controlling the operation of the robot and it is specially developed for being compatible with ROS. It is equipped with a STM32F7 chip that is based on ARM Cortex-M7. The OpenCR also includes many peripherals, in order to allow connection with and control of many devices. E.g. it is used to control dynamixels. Furthermore, it include the already discussed sensors, such as IMU's. OpenCR is shown in figure 6.3.

6.2.2 TREK1000

TREK1000 is an evalutation kit from Decawave. It is a good performance UWB (Ultra-Wideband) positioning system containing four units EVB1000 that can be used as anchors or tags. Each of them is equipped with a Decawave's wireless transceiver (DW1000 IEEE802.15.4-2011UWB compliant), a STM32F105 ARM Cortex M3, an USB peripheral, a display and a antenna (see figure 6.4). The system is designed for having a $\pm 10cm$ accuracy in static conditions and till $\pm 30cm$ with a moving target. Further information are included in table 6.1.

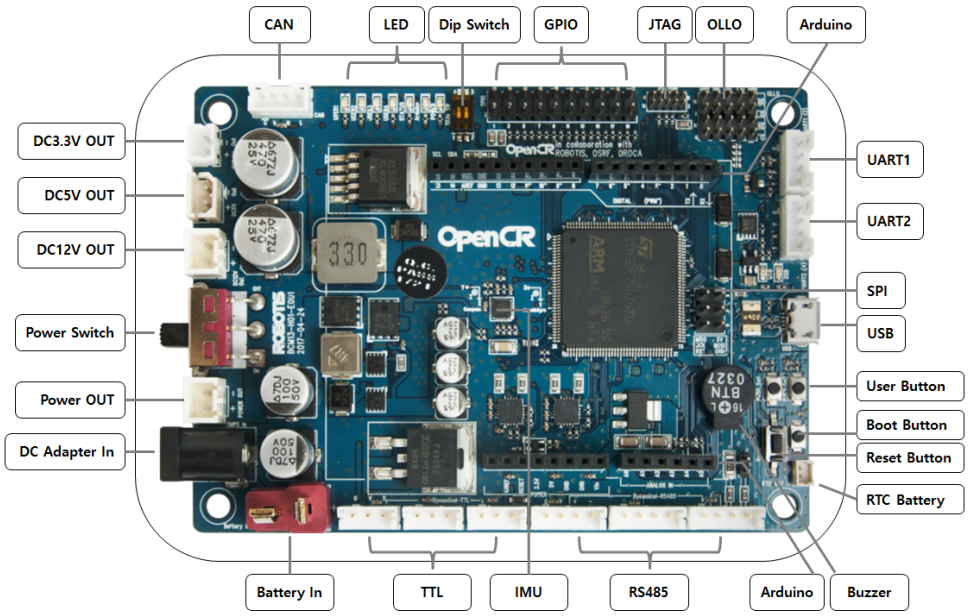


FIGURE 6.3: Labelled picture of the OpenCR.

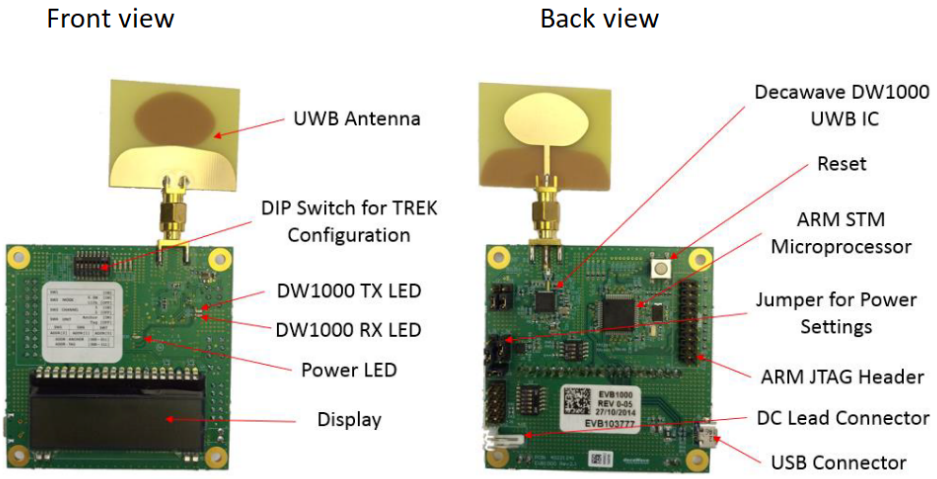


FIGURE 6.4: Scheme of the EVB1000 unit. On the left a the left front view is shown, on the right back one.

6.3 Software tools

Concerning the software side, that is, the control system, different tools are used. Basically any customized application is developed in python. Python is a widely used dynamic object-oriented programming language. it is especially suitable because there are a lot of resources available and it has a strong integration support with other languages and programs. Furthermore there is a good variety of libraries for practically any task and many application programming interfaces (APIs).

Board Dimensions	120x70 mm (including external antenna)
Weight	39 grams
Operating Band	3.5-6.5GHz 6 Channels (500 MHz wide) U.S. FCC mask compliant.
Center Frequency	Ch2: 3993.6 MHz Ch5:6489.6 MHz
Power spectral density	-41.3 dBm/MHz max
Antenna	External: WB002 Omni-directionalPlanar Antenna
Ranging Techniques	Pulsed Two-Way Time-of-Flight defined as TWR (Two-Way Ranging
Max Range	LOS: 290 m
Ranging precision	10 cm
Localization Technique	RTL software provided by manufacturer (see user manual for more infor-mation)
Network Protocols	TDMA
Max Positioning update rate (3 anchors, 1 tag)	3.57 Hz / 10 Hz (Depending on the operating mode)

Table 6.1: TREK1000 technical information

6.3.1 TensorFlow and Keras

TensorFlow is an end-to-end platform for machine learning (ML) that is open source. It was released back in 2015 by Google brain team, which is a Google project focusing on deep learning artificial intelligence research. It has complete ecosystem of tools, community resources, libraries that allows a state-of-the-art pushing in ML. It exists a TensorFlow API for python that allows performing a huge amount of different low and high level operations, e.g. the computation of gradients.

Keras is a TensorFlow high-level API (that is, it uses TensorFlow as back-end) that allows easy building and training of deep learning models. It is used both for research and production and it's very flexible.

6.3.2 ROS

For this work it is used Ubuntu, which is an open-source operating system that is based on Debian GNU/Linux. Ubuntu release 16.04 LTS is the one used. Concerning ROS, Kinetic Klame is the one coming with ubuntu 16.04 and hence it is used. The *Robot Operating System* (ROS) is a framework, a set of tools that can be used for providing the same functionalities, that can be found in any operating system, but on a cluster made of heterogeneous computers. ROS is used especially for robotics but has a lot of tools related to peripheral handling. It has five main principles:

- **Peer-to-Peer** (P2P): it has a P2P architecture that is coupled to a buffering system and to a lookup system (in ROS this is called 'master') that allows a direct dialogue between each components, both synchronously or asynchronously.
- **Multi-language**: ROS is not oriented to a particular language, and many of them can be used instead. For instance, for P2P, XML-RPC is used, which is available in a great number of languages.
- **Tools-based**: ROS has a microkernel design and it's not monolithic. It used many small tools to build and to run ROS components. Moreover each element executed and its execution is independent, that is, it does not affect others, making the system more flexible and robust with respect to a system based on centralised runtime environments.
- **Thin**: the idea is that drivers and algorithms should be standalone executables. This ensure re-usability and above all helps keeping size down. Moreover ROS takes code also from other open source projects, such as OpenCV, OpenRave, etc.
- **Free and open source**.

ROS is organized following a precise structure(see figure 6.5):

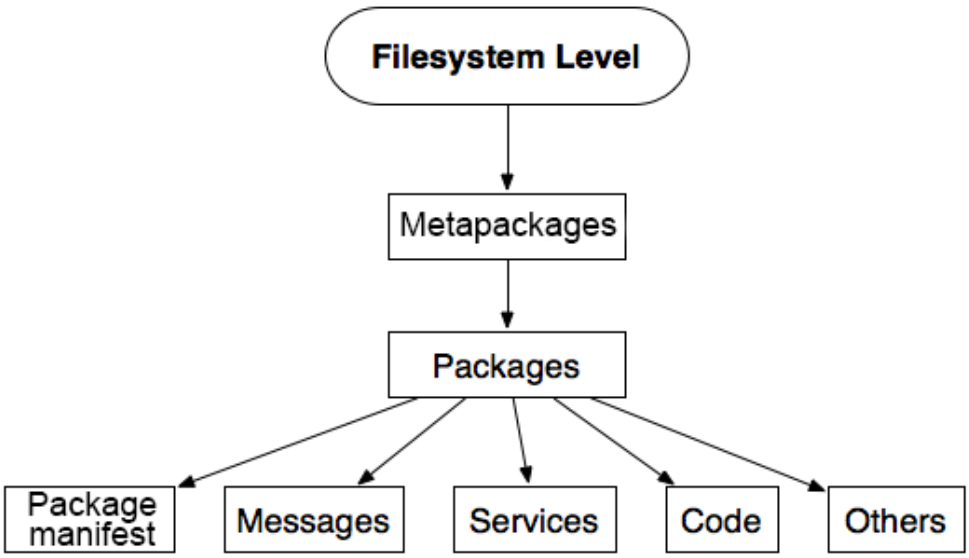


FIGURE 6.5: ROS code building organization.

- **Metapackages**: they are sets of packages, whose elements are grouped with same purpose.
- **Package manifest**: it contains information about the package, such as license, author, dependencies, and so on.

- **Packages**: ROS packages are the basic unit of ROS. A package contains nodes, configuration files, libraries, and so on.
- **Messages**: a ROS message define a type of information that is sent from one ROS process to another.
- **Services**: ROS service are some kinds of request/reply between processes.
- **Repositories**: ROS packages, hence ROS code, are maintained by using a **Version Control System** (VCS) such as subversion (svn), Git, etc.

For these work the following packages have been used:

- **turtlebot3**: it is a metapackage that can be used to handle turtlebot3 robots.
- **turtlebot3 machine learning**.
- **trek1000**: this package allows the handling of UWB anchors and the target for the localization.

Gazebo

Gazebo is a simulating tool especially suitable for robotics. It rich of roboticist's toolbox and makes possible to simulate and test design robots, algorithms, AI system training in high accuracy environments, performing regression testing. It gives access to dynamics simulation thanks to different physics engine, such as ODE, Bullet, DART, Simbody. Moreover a lot of robots model are provided, e.g. Pioneer2 DX, TurtleBot, iRobot Create and others can be built. Gazebo can be used remotely using socket-based messages. It's full of sensors such as laser range finders, 2D and 3D cameras, contact sensors, Kinect style sensors and many others. Furthermore there's an API that can be accessed via plugins. For this work Gazebo 7 is used.

6.4 Implementation

In this section it is described the structure of the system used during the training and how the training it's performed.

6.4.1 Robot setting

The overall systems is made of some elements: the robot (the burger) and its sensors (lidar, UWB sensors), the neural net that controls it.
Though the robot act in loop, data acquisition is actually the first step. Indeed, the neural network needs some information to be able to produce some reasonable outputs. However it doesn't receive completely raw data, hence to a data filtering step is required. Finally the neural network produces an output an the robot speed is modified. The cycle is then repeated again.

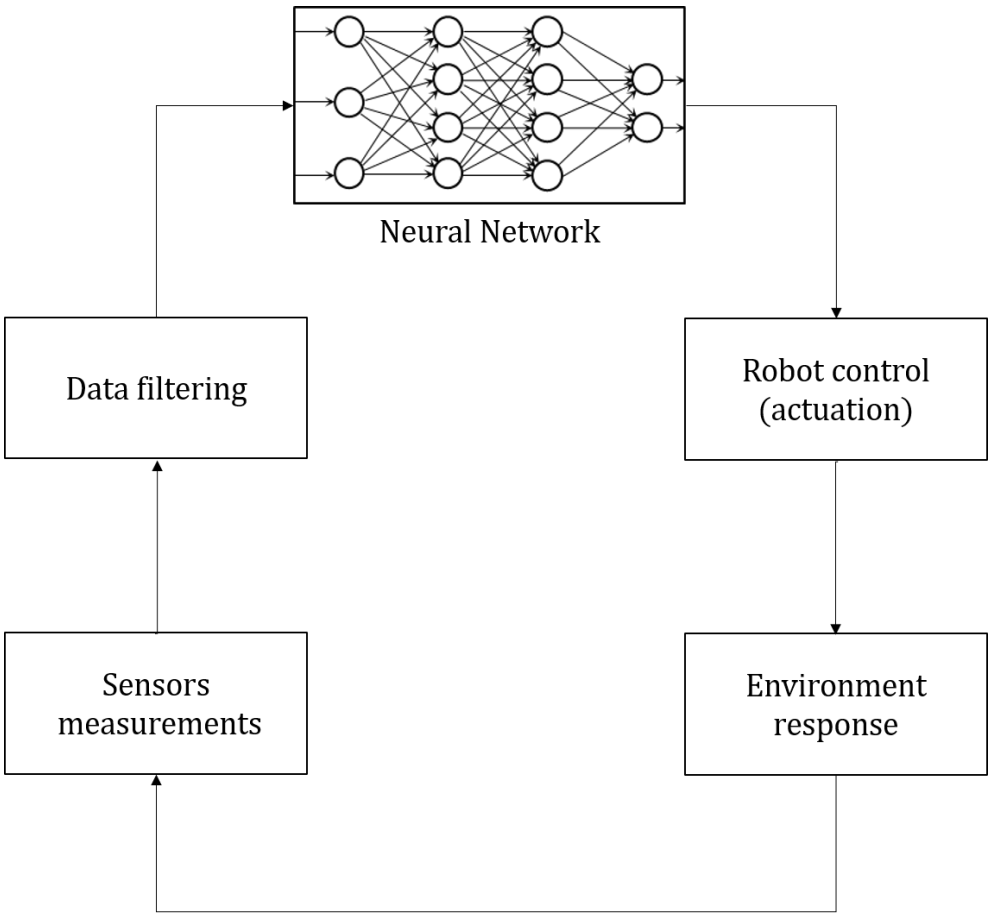


FIGURE 6.6: ROS code building organization.

Measurements

The agent, or rather the neural network, needs a representation of the state in order to compute an output. Hence some sensors are required. Briefly, raw data coming from different sensors are:

- **Lidar**: 359 elements array with the distance values from 0 to 359 degrees;
- **UWB system**: cartesian coordinates (x,y) in a global reference frame;
- **Odometry**: Robot pose with respect to burger reference frame.

In the following further information concerning these measurements are given.

The **lidar** measures the distances all around (see figure 6.7). The values are transmitted via ROS, hence they are published in a topic from a node. Among other information, there's a array with 359 elements, whose value are the distances corresponding to each measurements. The first element of the array refers to the null angle, or rather, right in front of the burger (this direction is shown as a blue segment in figure 6.7 (bottom)). The array goes than with a counter-clockwise verse.

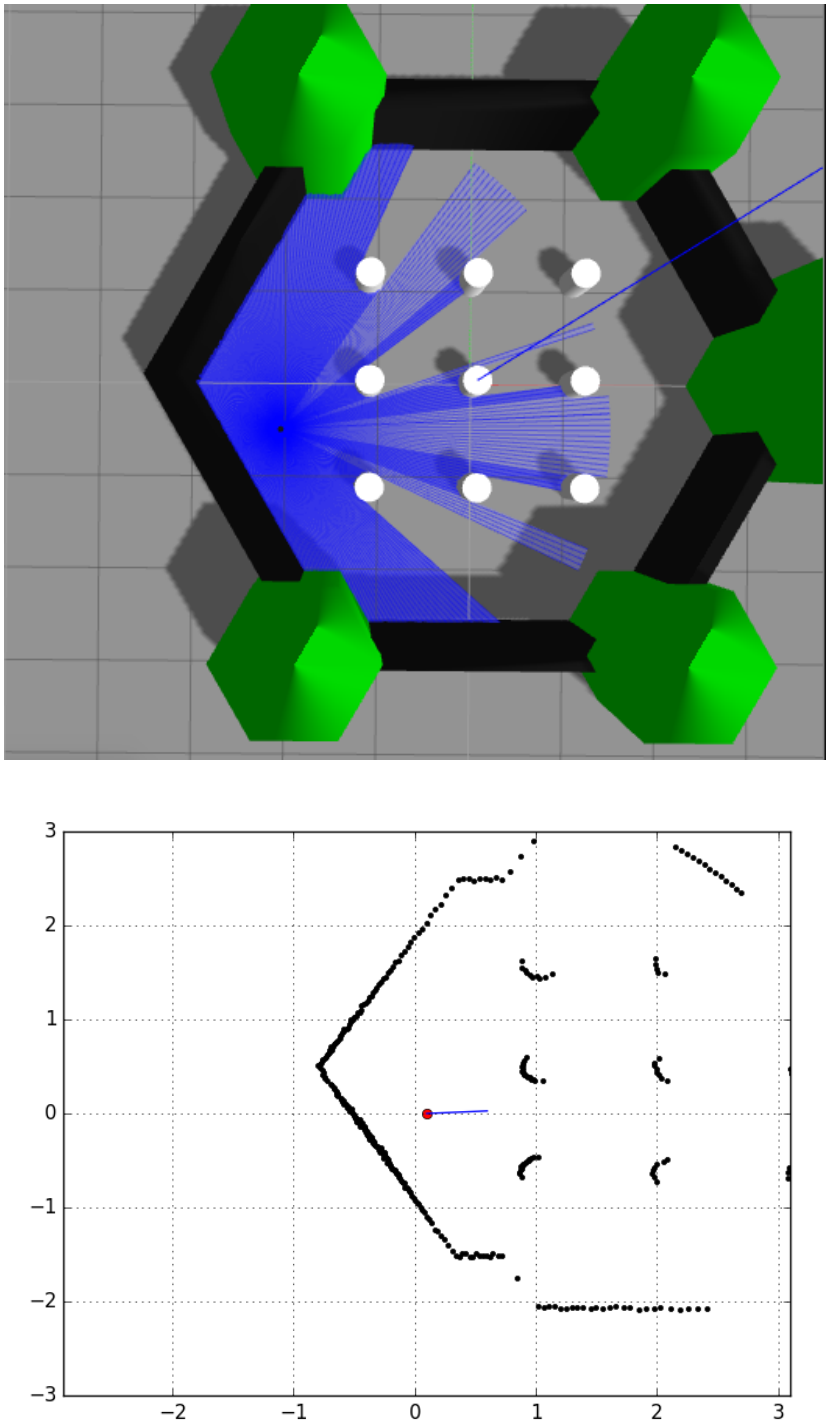


FIGURE 6.7: Top: screenshot from a Gazebo simulation, in which lidar range is also visible. Bottom: 2D cloud of lidar points. All the 359 points are shown.

TREK1000 anchors and the target on the burger provide the coordinates of the robot with respect to a global reference frame (see figure 6.8). As already explained in section dedicated to UWB, this system provide the coordinates of the target, computed from the distances from anchors and target and knowing anchors position in the global

reference frame.

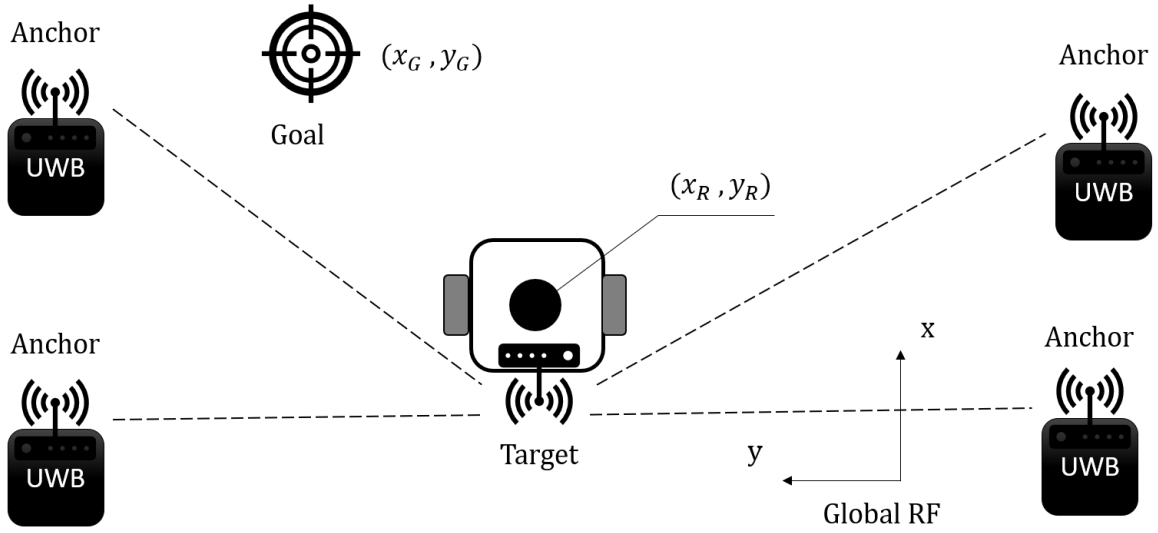


FIGURE 6.8: Scheme of the overall system.

Concerning **odometry**, related information are published thanks to OpenCR board. They are a published in a topic called */odom*. Both position and orientation are given, but while position is computed using robot model and by information coming from dynamixels, the orientation is provided by magnetometer. The latter indeed does not suffer from displacements, as the position.

Data processing

Not all data provided by sensors and openCR board are used. Furthermore they are slightly preprocessed:

Published data	⟶	**input for the network**
359 lidar points	⟶	60 (nearest per sector) lidar points
Odometry data	⟶	only orientation (yaw) is used
Odometry yaw and UWB coordinates	⟶	distance and angle with respect to the goal

TABLE 6.2

The number of lidar points is brought to 60 instead of 359 in order to reduce the number of the inputs for the network. This is done mainly to simplify the training phase and and, overall, to decrease network complexity and dimension. Before getting to the final setting, other solutions have been considered and tested: only 10 equally-spaced frontal points (FOV:180°), 30 equally-spaced all-around points (FOV:360°), 30 computed all-around points (FOV:360 °). The final solution include the aforementioned 60 points (FOV:360°). However, these yet not equally-spaced. The list of 359 measurements is split into 60 portions, or rather sectors. For each of this the minimum value is

chosen. This is done basically because the the nearer point is the most important, in terms of obstacle avoidance. For instance, when testing the system using 30 equally-spaced points, it was noticed that the robot had trouble in avoiding narrow objects. That's because with this solution, the object may not be detected at all. Instead, using the minimum of each sector, no information like that is lost. A scheme of what just described is shown in figure 6.9.

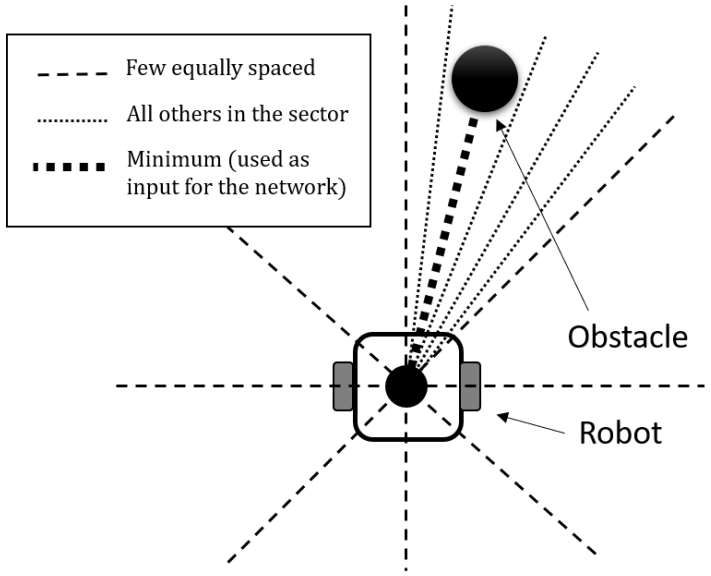


FIGURE 6.9: This example shows the result of using 6 points only. With 6 equally spaced measurements the obstacle would not be seen, while using the minimum value in each sector it can be detected.

In figure 6.10 it is shown what the neural network can see, in terms of lidar, in the same configuration as in figure 6.7.
Concerning odometry and UWB information, they are used to compute distance and orientation of the robot with respect to the target. A scheme is given in figure 6.11. They can be computed as follows. Angle between robot and goal is

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

then, using knowledge of the yaw γ (heading) can be computed as

$$\gamma = \begin{cases} \alpha - yaw, & if \quad -\pi \leq \alpha - yaw \leq \pi \\ \alpha - yaw - 2\pi, & if \quad \alpha - yaw > \pi \\ \alpha - yaw + 2\pi, & if \quad \alpha - yaw < -\pi. \end{cases}$$

Distance can be easily computed from coordinates:

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

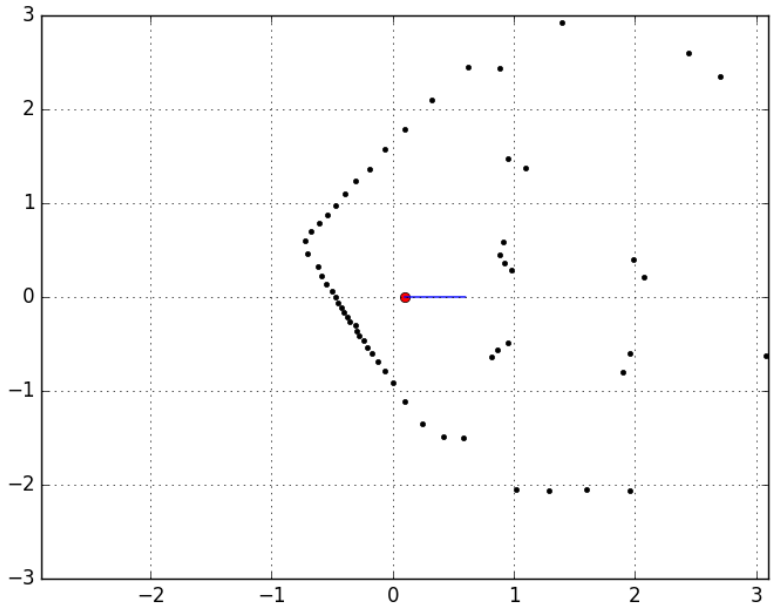


FIGURE 6.10: Lidar view with 60 points computed as minimum in each sector.

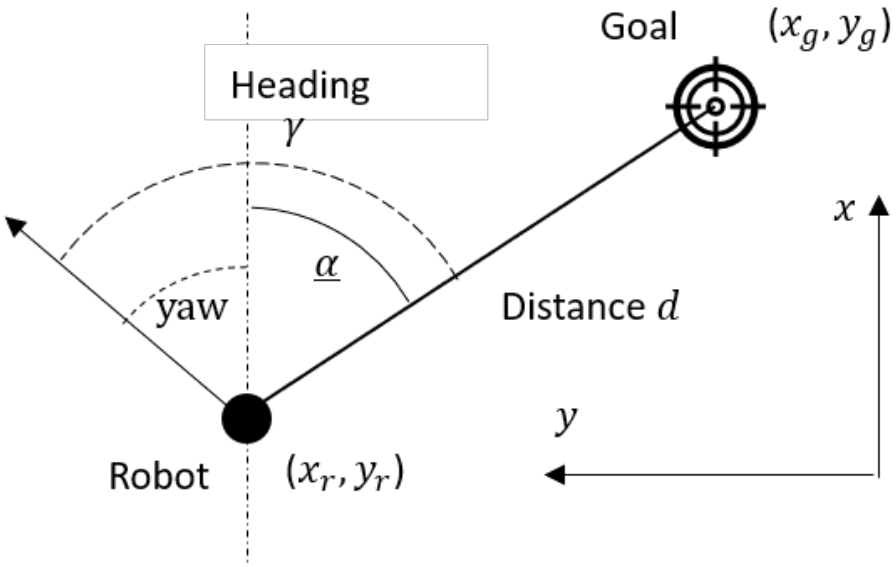


FIGURE 6.11: Scheme with robot and target with distance and heading representation.

Neural Network

As already explained in the section 4, the robot is controlled by a neural network that is called *actor*. Its structure is shown in figure 6.12. It accept an input of 62 elements that are basically lidar information, distance and heading. It has then some fully connected layer. Next, two different activation functions are used to get the output

values of linear and angular velocities. For the former, a *sigmoid* activation function is used, in order to have it in the range $[0, 1]$. Indeed the robot cannot have negative linear velocities. Concerning angular velocity, a *hyperbolic* function is used, to ensure a value in $[-1, 1]$. The output of these layer are then merged again and the network output is the robot action a_t Its components are then rescaled, by imposing maximum angular and linear velocities, that are:

- Max linear speed $0.2m/s$
- Max angular speed $1m/s$.

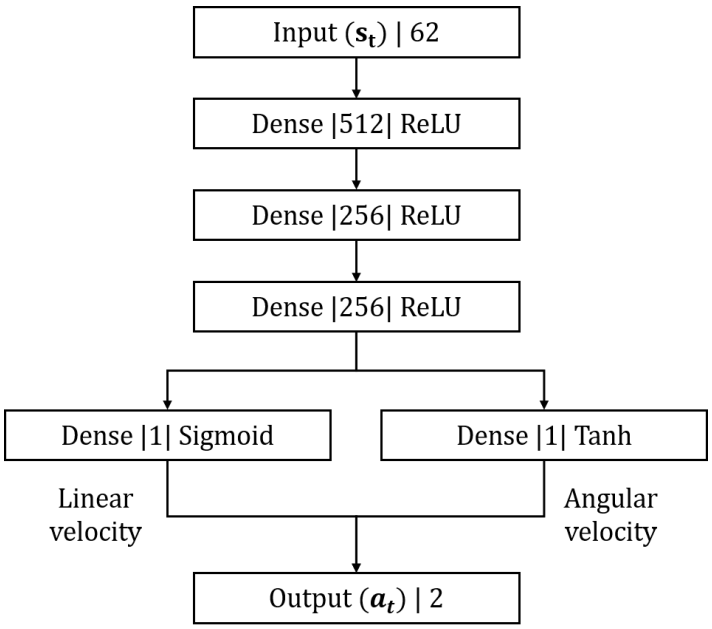


FIGURE 6.12: Structure of the actor network.

6.4.2 Training

The training phase is quite similar to the testing one, with the difference that some more elements are involved, such as simulated environment and critic network.

Training set up and environment

The training phase takes place in a simulation. The burger model is put into a environment. Different training scenarios has been taken into account, however in the end it has been used the one shown in figure 6.13.
In this scenario there are many different goals the robot has to reach. However, at the beginning there are fours different levels of difficulty, related the the four parts of the whole scenario. The communication system is practically the same as explained before, but UWB is not used, because odometry is trustful in short paths and in general in

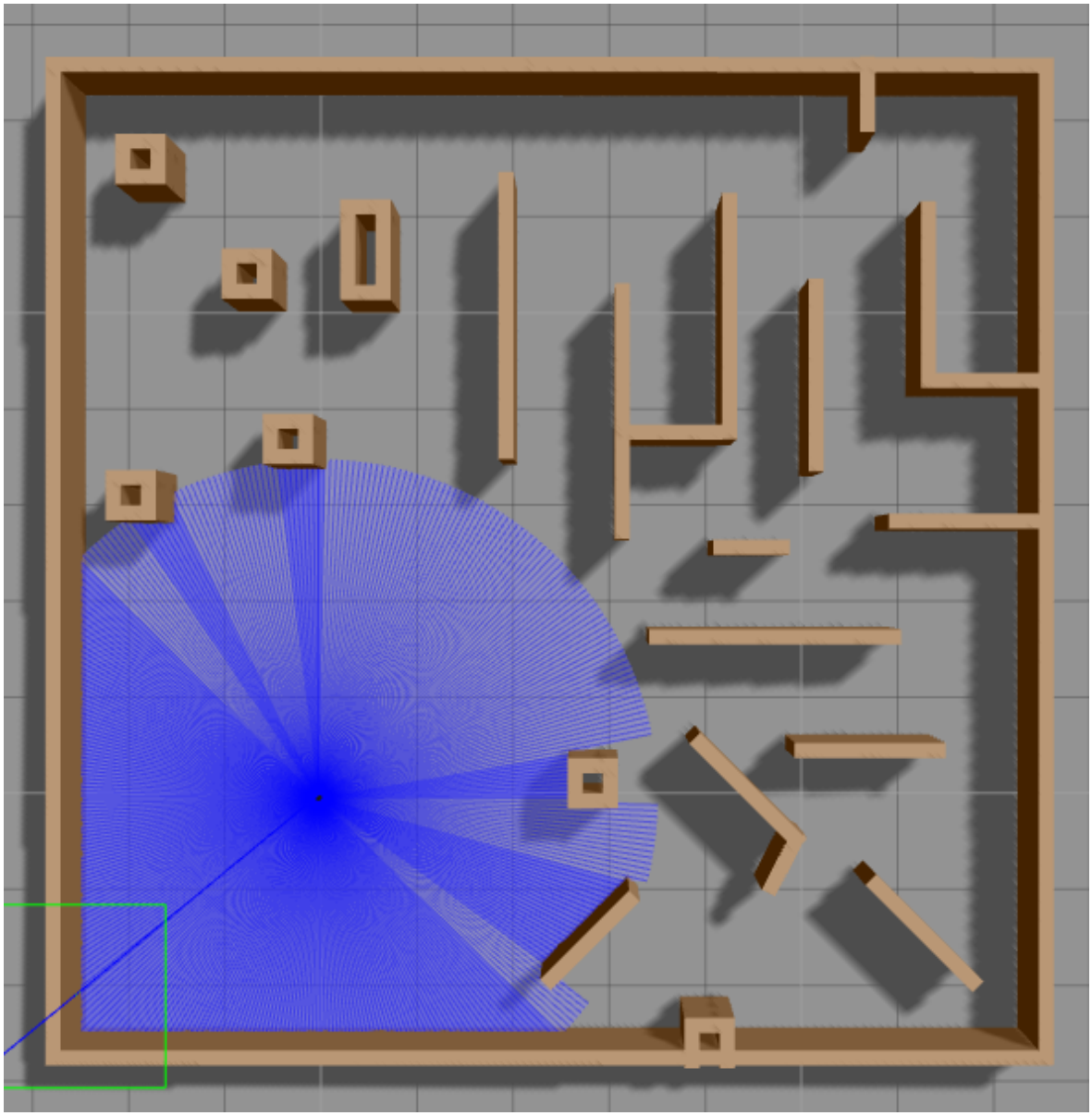


FIGURE 6.13: Scenario used for the training.

Gazebo and since there's no model of UWB sensors in Gazebo. The robot starts always from the same spawning point. A new goal is spawned when the agent gets to reach the previous one. When the robot collides with anything it is re-spawned at the start position and the episode ends. The episode end also after a certain amount of time, since there's a time-out.

Algorithm

The aforementioned neural network, called actor must be first trained. That means its parameters need to be adjusted in order to let it be a nice approximator. As already explained in section 4, this is done using DDPG. It include the use also of a *critic*, together with the actor. The former has the structure illustrated in figure 6.14.

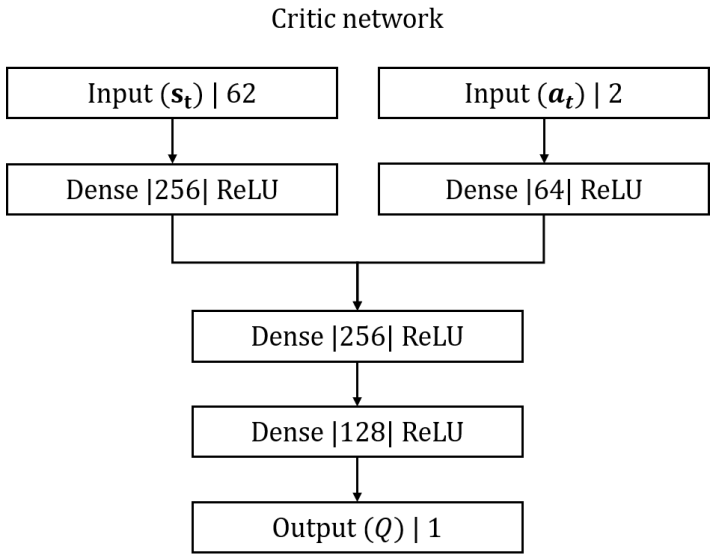


FIGURE 6.14: Structure of the critic network.

It receives as input the state and the action chosen by the actor and it gives as output an estimate of the related Q value. Critic network is trained using Adam optimization algorithm, exploiting TD difference, while actor parameters are trained using critic gradient.

6.4.3 Reward functions

Concerning the reward function, some main alternatives are considered. In all these cases however there's a common reward when reaching the goal and when colliding. First some used parameters are explicated:

$$\Delta d = d_{t-1} - d_t$$

$$d_r = distance\ rate = 2^{d_t/d_{t=0}}$$

$$h_R = heading\ reward = -(\omega_{t-1} \cdot \frac{1}{1.2 \cdot f} - heading)^2 + 1$$

where Δd is the distance from goal delta between current (d_t) and previous iteration (d_{t-1}), d_r is the distance rate, that is the ration between current distance (d_t) and distance at the beginning ($d_{t=0}$). h_R is a parameters that depend how the robot rotates with respect to the goal. ω is the angular speed, while f is the control frequency. The coefficient 1.2 is empirical.
At first an attempt was done using raw distance between robot and target:

$$R = \begin{cases} +1000 & \text{if goal is reached} \\ -200, & \text{if collision occurs} \\ \Delta d, & \text{else} \end{cases}$$

but it showed to not being able to converge. Further modification have led to improvements (such as bias adding) but not still acceptable results. Another function is the following:

$$R = \begin{cases} +1000, & \text{if goal is reached} \\ -200, & \text{if collision occurs} \\ 3 \cdot h_R \cdot d_r, & \text{else} \end{cases}$$

which proved to be effective. However in the end another function has been used, to avoid the use of the distance rate (d_r):

$$R = \begin{cases} +1000, & \text{if goal is reached} \\ -200, & \text{if collision occurs} \\ 3 \cdot h_R \cdot 10 \cdot |\Delta d|, & \text{else} \end{cases}$$

where the coefficient 10 is due to equilibrate the reward with respect to the other cases (goal and collision).

Hyperparameters and settings

Overall, many parameters have been modified and changed, not only the reward function. In table 6.3 the final values are shown.

Parameter	Value
Algorithm hyperparameters	
starting epsilon	1
minimum epsilon	0.05
epsilon decay	0.998
learning rate	0.00025
discount factor	0.99
sample size	64
batch size	64
target network update	2000
deque memory maxlen	1000000
Robot settings	
lidar points	60
ctrl period	0.33
maximum angular speed	$1 rad/s$
maximum linear speed	$0.2 m/s$
Simulation settings	
time step	$0.0035 s$
max update rate	$2000 s^{-}1$
timeout	$250 s$ (in sim. time)

Table 6.3: Hyperparameters and settings.

As already written, the robot must face the exploration-exploitation dilemma. This is done by using a noise on policy, to allow random policies. The agent have a ϵ probability to choose a random action. This epsilon decrease over training (thanks to *epsilon decay*) till a minimum value *minimum epsilon*, to ensure some exploration. However there's no such stochastic chance when testing, since epsilon is set to zero. Parameters such as *learning rate* and *discount factor* are related to the methods used to perform the shift in parameters, e.g. the Adam, along with *batch size*. *Sample size* is simply the amount of records that are used to perform a fitting, at each iteration. The *target network update*, as explained in section 4 is a parameters that establishes how often target network (used to increase convergence) are updated. *Deque memory maxlen* give the length of the memory in which experience are stored for training. This type of memory discard older records when reaches the maximum number of element.

Main settings concerning the robot platform are related to the number of lidar measurements to be sent to the network, the control period (please note that this frequency was used in training to speed it up; in testing phase instead it runs at 30 Hz), that is the reciprocal of the frequency.

Concerning simulation settings, *time step* and *max update rate* define simulation accuracy and speed, indeed their product gives the real time factor, that for the simulation was around 7 (that is, the simulation were run seven times faster than real time).
In the following some learning curves are shown. Figure 6.15 illustrates a learning curve that is converging to a negative value. In this case the robot did not learn how to reach the goal, instead it started to hanging around near the spawn point, in order to avoid any obstacle. This was obtained using a reward function similar to the first of the three reported in the previous pages.

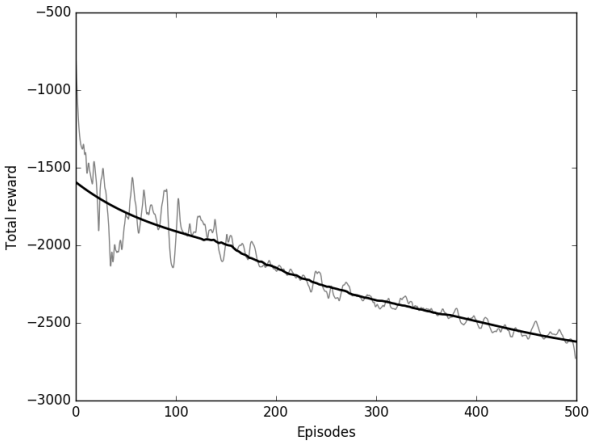


Figure 6.15: Learning curve in case of wrongful convergence. The simulation is stopped to avoid useless work.

In figure 6.16 it is illustrate a learning curve when the algorithm succeeds. This was obtaining using the second type of reward function.

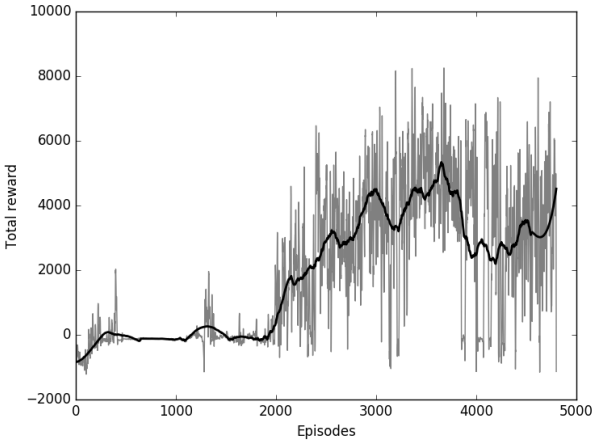


FIGURE 6.16: Learning curve when the algorithm succeeds, with the second reward function.

Finally in figure 6.17 it is shown the learning curve of the agent that used for the testing phase. After approximately 3000 episode the maximum reward is obtained.

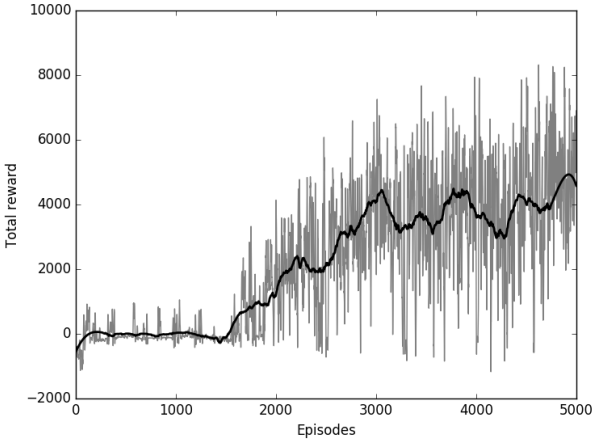


FIGURE 6.17: Learning curve when the algorithm succeeds, with the third reward function.

Chapter 7

Results and Conclusions

The overall system was finally tested in a real environments. This chapter include information about each environment, the metric that was used and the results of both DRL agent and human beings.

7.1 Test Metrics

To evaluate the performance of the autonomous mobile robot there's the need for metrics, that is, a set of performance measurements to which a quantitative value can be given. Here a set of six parameters is used.

- **Success**. The first parameter to take into account is the success of the robot. The value is of course binary: 1 (*yes*) when the robot manage to reach the goal and 0 (*no*) when it does not. The other judgement parameters are hardly meaningful without this one.
- **Time spent**. The time-interval goes from when the task starts until the robot reaches the goal (with the given tolerance). Given the test environments, when this value exceeds $120s$ it is considered to have failed.
- **Path length**. This value is computed considering the total length of the path.
- **Collisions**. The number of collision is collected and used to give penalties..
- **Acceleration**. Linear and angular acceleration are taken into account to evaluate the smoothness of the control.
- **Jerk**. The jerk is the derivative over time of the acceleration. Here linear and angular jerk are considered, again for comparing the smoothness of the different controls.

7.1.1 Optimality

Of the set just described, some elements can be referred to as optimality factors. For instance, *time spent* and *path length* are a direct way to compare algorithms or optimality performances . In mobile robotics and path finding they are usually considered as the most important. However it is possible that an agent with a little precautionary policy eventually collides with the environment, especially in a non-static one. Hence also the number of collisions has been taken into account.

7.1.2 Smoothness

Another element to take into account, when talking about navigation (especially for service robotics) is the smoothness. That's clearly related with the increase in human-interaction and it can be considered the most important factor to take into account in some application, e.g. those where the agent has to takes a human somewhere, that is, the navigation must be as comfortable as possible. Acceleration and jerk are usually used as evaluation parameters (see [21],[22]) in many different ways, e.g. as integral function or just considering the peaks. However, in these work, both velocities, accelerations and jerks were computed as derivative from position, leading to low-quality data. Nonetheless data collected for the DRL agent and for people has followed the same procedure; hence it is believed they can be used for comparison purposes. Smoothness is evaluated considering the RMS (Root Mean Square) values of linear and angular accelerations and jerks.

7.1.3 Ultra-wideband

Concerning Ultra-Wideband, it was not possible to have analytical data to evaluate its goodness. However, it plays an important role, that is discussed in the next section.

7.2 Environments and set-up

During the testing phase, six different stages were taken into account (numbered as $0, 1, 2, 3, 4, 5, 6$). However only four of them $(1, 3, 4, 5)$ were considered in the end and here used. In each stage the robot had to go from a point to another. In stages 4 and 5 the robots had more tasks to accomplish. The coordinates of the points have been measured with the total station, in the same reference frame as the UWB anchors. The positioning of the robot has been made by hand, however the related error is considered negligible to respect the used tolerances.

7.2.1 Stage 1

The first stage is a non-static simple environment. The map is shown in figure 7.1. The agent starts from $S(0.00, 0.00)$ (starting point) and has to reach the goal $G(4.45, 0.02)$

with a tolerance ($t = 0.2$). After $8s$, which has been seen to be the best moment to interrupt the navigation, a person step ahead of the robot, making it modify its route.

FIGURE 7.1: Scheme of environment 1.

7.2.2 Stage 3

The third environment is again a non-static environment. Start and goal have the same coordinates as in stage 1 ($S(0.00, 0.00)$,$G(4.45, 0.02)$). The environment provides only a way tor reach the goal. When the robot is about 0.4 away from this passage (see figure 7.2) a person steps there and closes it for $3s$, then goes away.

FIGURE 7.2: Scheme of environment 3.

7.2.3 Stage 4

Stage 4 involves a maze-like static environment (see 7.3) in which the agent has to perform three tasks (called a,b, and c).

(a) from $S(0.00, 0.00)$ to $G(4.45, 0.02)$

(b) from $S'(4.45, 0.02)$ to $G'(1.86, -0.21)$

(c) from $S''(1.86, -0.21)$ to $G''(1.645, 1.654)$

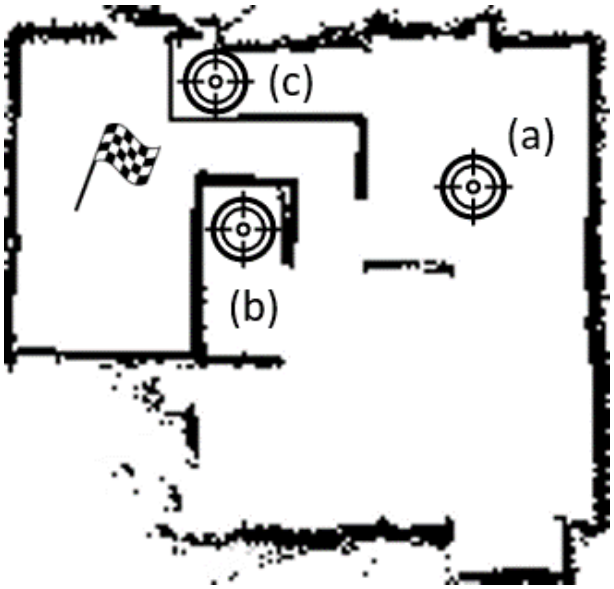


FIGURE 7.3: Scheme of environment 4.

7.2.4 Stage 5

The last environment is maze-like and non-static. Moreover it include the presence of people that disturb the path of the robot. The route done by the people is always the same. Also stage 5 involves three tasks. The point are the same but the order changes.

(a) from $S(0.00, 0.00)$ to $G(1.86, -0.21)$

(b) from $S'(1.86, -0.21)$ to $G'(4.45, 0.02)$

(c) from $S''(4.45, 0.02)$ to $G''(1.645, 1.654)$

7.2.5 Human interface

As already commented, people were used to compare performances of the used DRL algorithm. In the following information about the interface is given. Each person was led in a room different from the one containing the environment, in order to avoid any

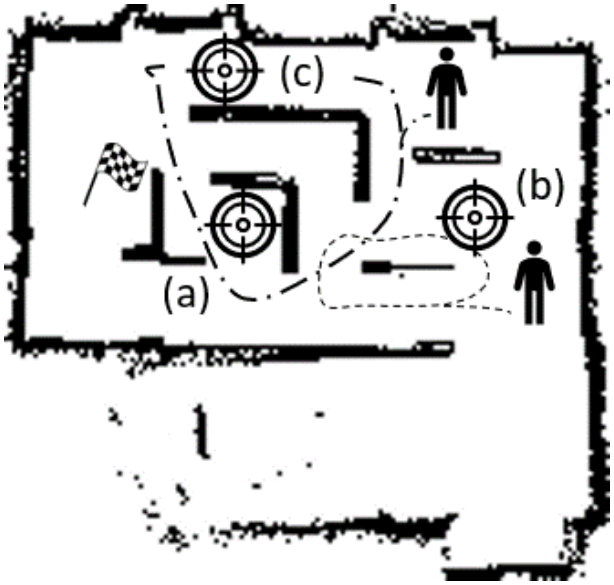


FIGURE 7.4: Scheme of environment 5.

of them to see the environment itself. They could hence operate remotely having the same information as the DRL agent: lidar information, position provided by UWB technology, goal position. What is seen by any people is shown in figure 7.5.

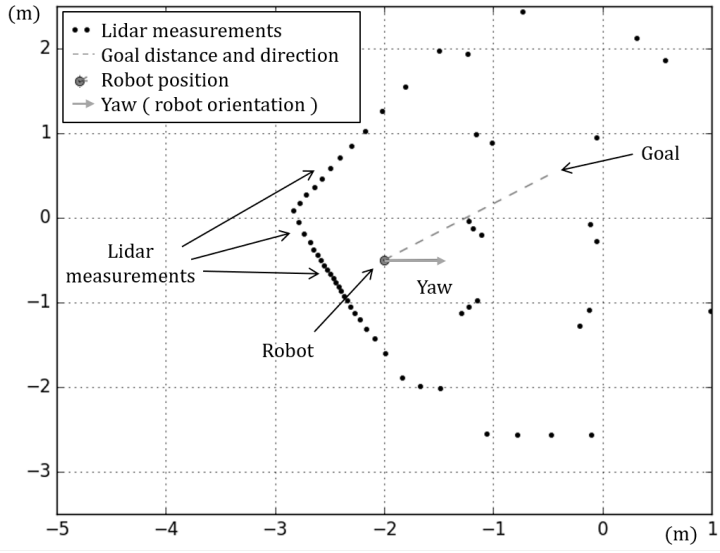


FIGURE 7.5: Information-interface for human control.

This representation of the state, or rather of the map, had a refresh rate equal to that of the lidar, hence $5Hz$. Concerning the control, the human agent could modify linear and angular velocity much less freely than the DRL agent. The used device is the keyboard, in particular the four keys *WASD* are used to control linear speed (W and S) and angular one (A and D).

- *Linear speed*: it could be increased with a step of $0.5m/s$ (in control, though

turtlebot acceleration may not have been sufficient) till a peak of $0.22m/s$. Pressing S key made the robot stop. Human could set negative speed with key X, though it was never useful. The linear speed was set, that means that even if the key was not pressed any longer the robot continued going at that speed.

- *Angular speed*: the human operator could control the angular speed using A and D. However pressing those keys sent a fixed control command, that was a speed of module $1.7rad/s$ (direction depending on the key). However this time of command was just instantaneous, that means, the angular speed was again set to zero one the key was released.

Each person was allowed to have some training before doing the tests, in order to learn how to manage the robot, also to understand its agility and dimensions.

7.3 Comparison

In the following the results of human agents and the DRL one are exposed, in order of stage. For each stage, data are condensed in tables, and only some trajectories are reported here. However all the graphs can be observed in the appendix.

7.3.1 Stage 1

While for people there was test only, the DRL were tested five times. The same procedure was followed also for stages 3 and 4. In the following a summary table is presented (see table 7.1).

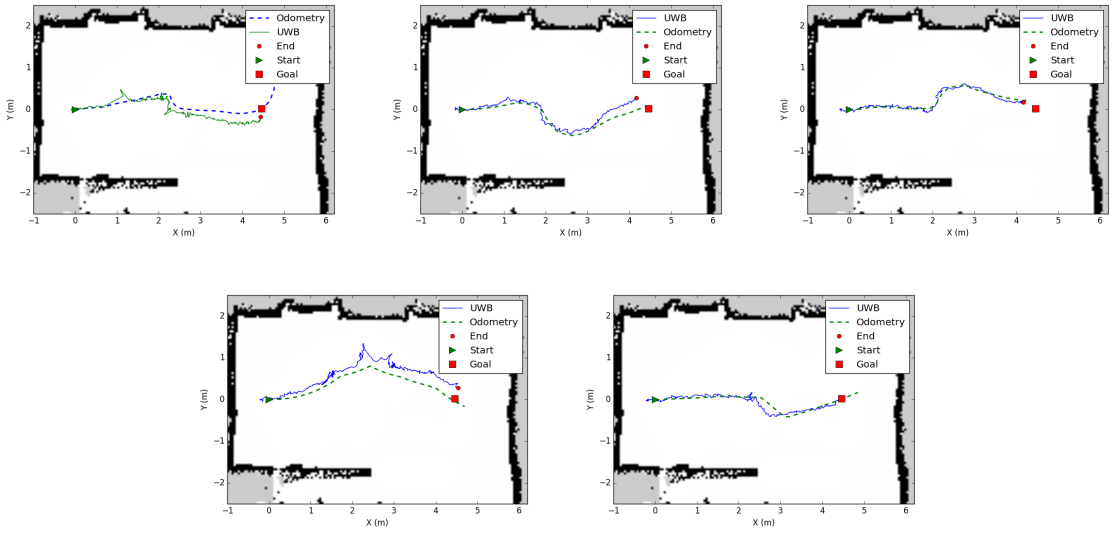


FIGURE 7.6: Trajectories of the five agents in stage 1: DRL (top-left), person 1 (top-centre), person 2 (top-right), person 3 (bottom-left), person 4(bottom-right).

STAGE 1	DRL	Person 1	Person 2	Person 3	Person 4
Success (y/n)	y	y	y	y	y
Time (s)	$26 \div 33$	28	25	40	27
Path length (m)	5.2	4.9	4.6	5.1	5.1
Number of collisions	0	0	0	0	0
Avg lin. speed (m/s)	0.16	0.16	0.16	0.11	0.17
RMS ang. speed (rad/s)	0.29	0.26	0.29	0.22	0.27
RMS lin. acc. (m/s^2)	0.36	0.37	0.39	0.32	0.35
RMS ang. acc. (rad/s^2)	3.8	2.1	2.1	1.8	2.4
RMS lin. jerk (m/s^3)	14.5	6.6	7.6	6.0	6.5
RMS ang. jerk (rad/s^3)	73	36	38	39	44

TABLE 7.1: Summary table of stage 1.

In the first stage DRL performances weresimilar to human operators both in terms of time and path length (see figure 7.6). However it shows to have a less smooth navigation than the operators (see values of acceleration and jerk). The result are more or less the same among the other stages.

7.3.2 Stage 3

The trajectories of stage 3 can be seen in the appendix. They are not reported for space reasons.

STAGE 3	DRL	Person 1	Person 2	Person 3	Person 4
Success (y/n)	y	y	y	y	y
Time (s)	$44 \div 54$	35	44	60	59
Path lentgh (m)	6.8	6.0	5.9	6.6	5.8
Number of collisions	0	0	0	0	0
Avg lin. speed (m/s)	0.13	0.16	0.12	0.10	0.11
RMS ang. speed (rad/s)	0.48	0.23	0.53	0.33	0.34
RMS lin. acc. (m/s^2)	0.33	0.38	0.33	0.32	0.32
RMS ang. acc. (rad/s^2)	4.5	1.9	2.3	1.9	1.9
RMS lin. jerk (m/s^3)	13.9	7.9	5.9	12.1	6.8
RMS ang. jerk (rad/s^3)	74	46	33	42	38

TABLE 7.2: Summary table of stage 3.

7.3.3 Stage 4

Stage 4, though involving a static environment, has proved to be the toughest for both humans and the DRL agent. However three out of the four people have manage to

complete all the task (the other one has spent a huge amount of time and been considered unsuccessful). On the other hand the DRL agent has not manage at all to complete the last task ((c)) and it has not been for a matter of time. The trajectories can be observed in the appendix.

STAGE 4 - a	**DRL**	**Person 1**	**Person 2**	**Person 3**	**Person 4**
Success (y/n)	y	y	y	y	y
Time (s)	$39 \div 41$	39	33	32	59
Path length (m)	5.3	5.9	5.4	6.0	6.9
Number of collisions	0	0	0	0	**1**
Avg lin. speed (m/s)	0.13	0.13	0.14	0.16	0.13
RMS ang. speed (rad/s)	0.56	0.44	0.34	0.38	0.46
RMS lin. acc. (m/s^2)	0.35	0.35	0.38	0.40	0.35
RMS ang. acc. (rad/s^2)	4.1	2.1	2.1	2.0	2.2
RMS lin. jerk (m/s^3)	14.3	6.6	8.4	14.8	6.1
RMS ang. jerk (rad/s^3)	73	41	40	50	34

Table 7.3: Summary table of stage 4, task a.

STAGE 4 - b	**DRL**	**Person 1**	**Person 2**	**Person 3**	**Person 4**
Success (y/n)	y	y	y	y	y
Time (s)	$27 \div 31$	26	25	69	24
Path lentgh (m)	4.0	3.9	3.7	8.3	3.8
Number of collisions	0	0	0	**2**	0
Avg lin. speed (m/s)	0.13	0.14	0.17	0.1	0.15
RMS ang. speed (rad/s)	0.52	0.44	0.41	0.38	0.48
RMS lin. acc. (m/s^2)	0.35	0.39	0.38	0.40	0.35
RMS ang. acc. (rad/s^2)	4.1	2.0	2.3	1.9	2.1
RMS lin. jerk (m/s^3)	14.8	8.8	6.7	6.4	6.7
RMS ang. jerk (rad/s^3)	72	47	39	44	31

Table 7.4: Summary table of stage 4, task b.

7.3.4 Stage 5

The trajectories for each agent can be observed in the appendix.

7.3.5 Overall comparison

In the following an overview of the presented data is provided. First, in figure 7.7 the total amount of time spent for each stage and task is shown. Task c of stage 4 is missing, since the time spent for the agent is undefined, not having it completed the

STAGE 4 - c	DRL	Person 1	Person 2	Person 3	Person 4
Success (y/n)	n	y	y	n	y
Time (s)	> 120	100	80	> 120	113
Path lentgh (m)	$\sim$	6.0	5.9	6.6	5.8
Number of collisions	0	0	0	**1**	0
Avg lin. speed (m/s)	$\sim$	0.13	0.16	0.08	0.18
RMS ang. speed (rad/s)	$\sim$	0.49	0.45	0.39	0.39
RMS lin. acc. (m/s^2)	$\sim$	0.38	0.35	0.27	0.38
RMS ang. acc. (rad/s^2)	$\sim$	2.2	2.1	2.0	2.2
RMS lin. jerk (m/s^3)	$\sim$	8.5	7.6	5.1	10.9
RMS ang. jerk (rad/s^3)	$\sim$	48	40	32	44

Table 7.5: Summary table of stage 4, task c.

STAGE 5 - a	DRL	Person 1	Person 2	Person 3	Person 4
Success (y/n)	y	y	y	y	y
Time (s)	49	48	29	40	80
Path lentgh (m)	5.6	6.5	4.1	3.9	10.6
Number of collisions	0	0	0	0	0
Avg lin. speed (m/s)	0.12	0.10	0.13	0.09	0.12
RMS ang. speed (rad/s)	0.58	0.37	0.48	0.45	0.55
RMS lin. acc. (m/s^2)	0.32	0.34	0.35	0.27	0.33
RMS ang. acc. (rad/s^2)	4.6	1.6	2.0	2.3	2.5
RMS lin. jerk (m/s^3)	13.0	6.6	6.3	4.7	6.3
RMS ang. jerk (rad/s^3)	74	28	30	34	50

Table 7.6: Summary table of stage 5, task a.

STAGE 5 - b	DRL	Person 1	Person 2	Person 3	Person 4
Success (y/n)	y	y	y	y	y
Time (s)	28	43	33	45	39
Path lentgh (m)	4.2	2.7	4.3	5.1	5.6
Number of collisions	0	0	0	0	0
Avg lin. speed (m/s)	0.14	0.07	0.12	0.10	0.13
RMS ang. speed (rad/s)	0.49	0.69	0.41	0.35	0.41
RMS lin. acc. (m/s^2)	0.34	0.30	0.34	0.31	0.34
RMS ang. acc. (rad/s^2)	4.4	2.6	2.3	2.2	2.5
RMS lin. jerk (m/s^3)	14.0	6.2	5.7	5.4	11.8
RMS ang. jerk (rad/s^3)	77	47	34	33	54

Table 7.7: Summary table of stage 5, task b.

task. For DRL agent average values are used. Considering the overall result it is well placed.

Figure 7.8 shows graphically the total length travelled by each agent through all levels. It can be noticed that DRL agent's performances are placed among the other ones.

STAGE 5 - c	DRL	Person 1	Person 2	Person 3	Person 4
Success (y/n)	y	y	y	y	y
Time (s)	35	49	40	74	30
Path lentgh (m)	7.6	5.9	5.9	7.1	5.1
Number of collisions	0	0	0	0	0
Avg lin. speed (m/s)	0.12	0.11	0.14	0.09	0.16
RMS ang. speed (rad/s)	0.60	0.50	0.47	0.39	0.34
RMS lin. acc. (m/s^2)	0.33	0.35	0.37	0.29	0.36
RMS ang. acc. (rad/s^2)	4.4	2.3	2.0	2.1	2.4
RMS lin. jerk (m/s^3)	13.7	6.7	6.5	5.2	12.1
RMS ang. jerk (rad/s^3)	73	34	32	35	66

Table 7.8: Summary table of stage 5, task c.

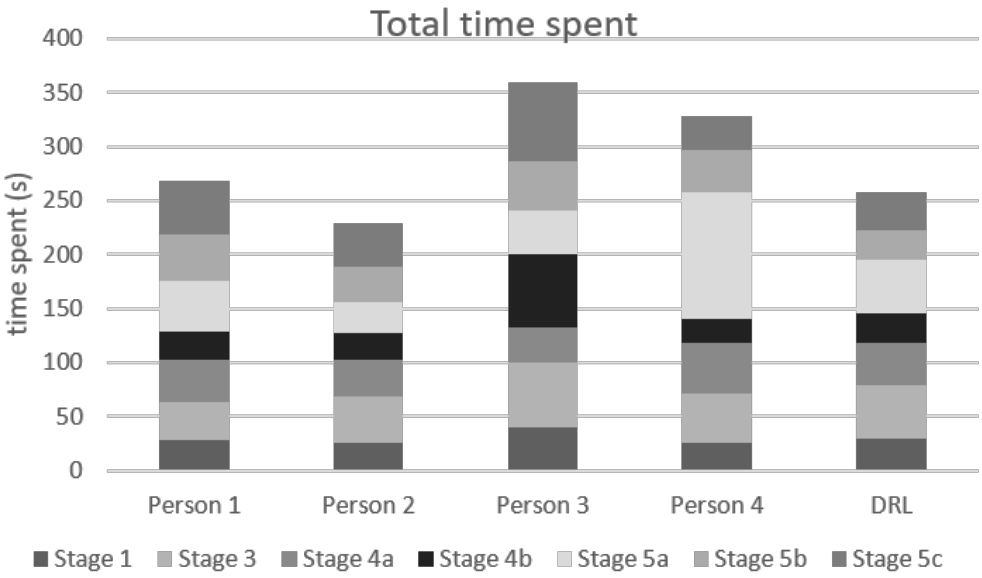


Figure 7.7: Graph reporting the total time spent for each agent in each task. Task c of stage 4 is missing, since it has not been accomplished by all agent.

In terms of optimality it can be noticed that the DRL agent did well. Moreover it acted in all the tasks without ever colliding, while person 1, 3 and 4 did (see stage 4 and 5). On the other hand, DRL agent did not managed to accomplish task c of stage 4, being stacked in a local maze-local minimum. Furthermore,in terms of smoothness, human agents provided a much more smooth movement. It must be noticed however that the control given them is much simpler ans it intrinsically smoother. In figure 7.9 a comparison between smoothness-related values is shown. The values of human agents are averaged.

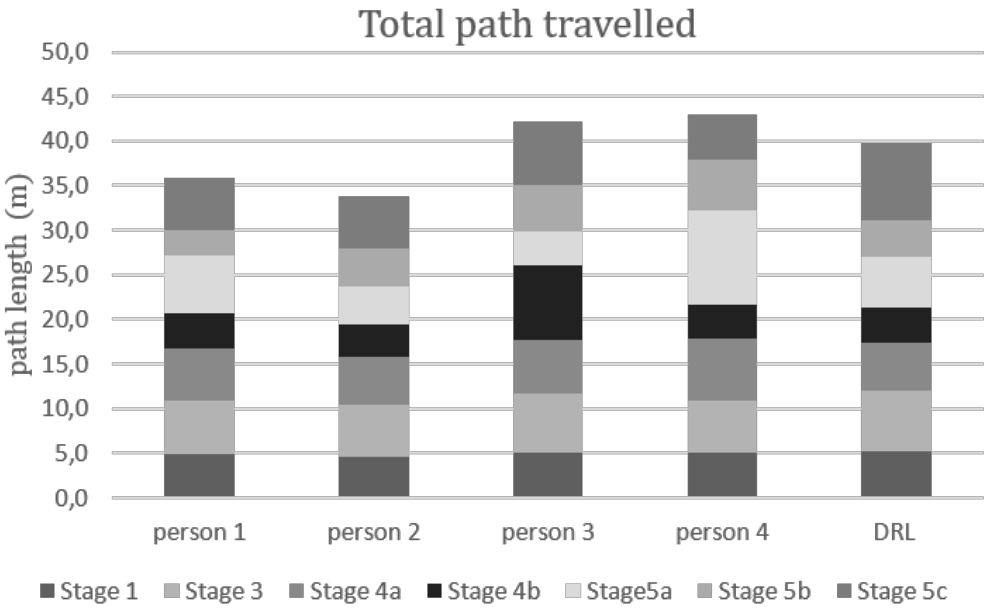


FIGURE 7.8: Graph reporting the total path length for each agent in each task. Task c of stage 4 is missing, since it has not been accomplished by all agent.

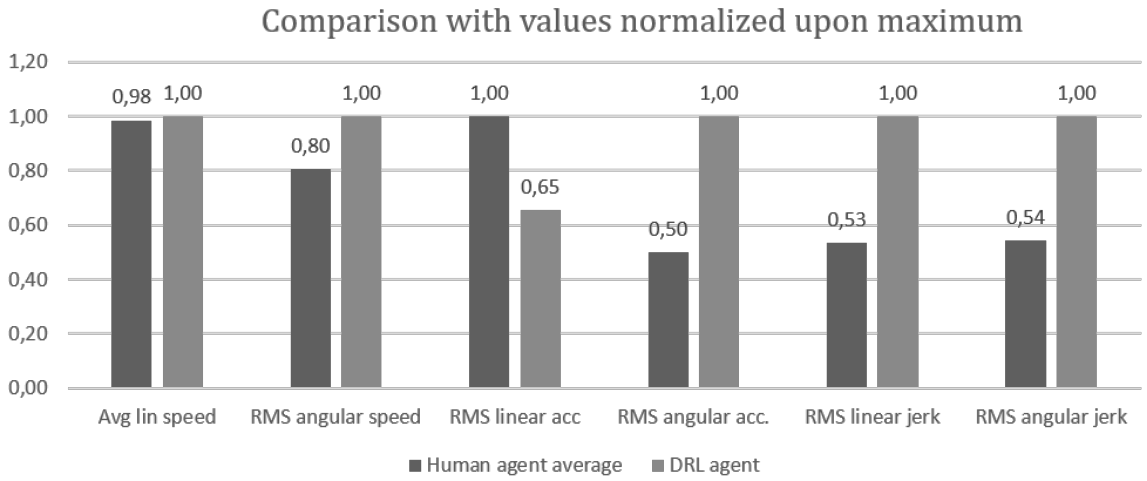


FIGURE 7.9: The diagram shows normalized (upon the maximum, per category) and averaged values of linear speed and RMS value of angular speed, linear and angular acceleration, linear and angular jerk.

7.3.6 Upon Ultra-wideband

The use of ultra-wideband has not been evaluated or compared quantitatively. A discussion upon its advantages with respect to odometry would have been of course trivial. Hence what must be noticed is that it has been usable and it complied with the localization requirements, at this stage. Figure 7.10 shows two examples of UWB trace with respect to odometry one. The latter can be considered generally trustful when

short distance are taken into account and hence can be used as reference for seeing how the UWB trace follows it. However, the example on the right shows the importance of having a robust system when collision occur.

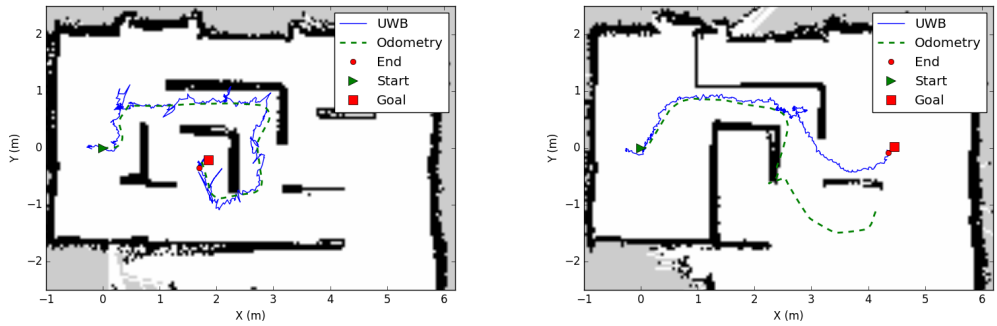


FIGURE 7.10: View of two examples of UWB and odometry trajector.

7.4 Conclusions and future work

The DRL agent has shown to have a behaviour similar to those of the people that participated to tests. This is believed to be a success, considering the challenge taken into account. Though the agent has known weaknesses, such as the total lack of a memory, it already represents a good result.
Ultra-wideband technology, even at this almost-embryonic stage, has shown to be a usable and valid tool for localization. Moreover, the fact it is quite noisy helps understand the robustness of the DRL agent. The overall system is considered to be complying with the expectation, both in terms of performances and low-computational resources required, and it is indeed a baseline for future works.

Next steps will be:

- Adding a feature that can allow the agent to overcome failure situation (see stage 4, task c), such as a memory;
- The improvement of UWB filtering;
- The improvement of DDPG algorithm, e.g. by adding some known feature that could help improving performances;
- Use of computer vision to improve state representation quality;

Bibliography

[1] M. Mohammadi et al. "Semisupervised Deep Reinforcement Learning in Support of IoT and Smart City Services". In: *IEEE Internet of Things Journal* 5.2 (Apr. 2018), pp. 624–635. DOI: 10.1109/JIOT.2017.2712560.

[2] Lei Tai, Giuseppe Paolo, and Ming Liu. "Virtual-to-real Deep Reinforcement Learning: Continuous Control of Mobile Robots for Mapless Navigation". In: *CoRR* abs/1703.00420 (2017). arXiv: 1703.00420. URL: http://arxiv.org/abs/1703.00420.

[3] Andrey Gavrilov and Artem Lensky. "Mobile Robot Navigation Using Reinforcement Learning Based on Neural Network with Short Term Memory". In: vol. 6838. Aug. 2011, pp. 210–217. DOI: 10.1007/978-3-642-24728-6_28.

[4] Somil Bansal et al. "Combining Optimal Control and Learning for Visual Navigation in Novel Environments". In: *CoRR* abs/1903.02531 (2019). arXiv: 1903.02531. URL: http://arxiv.org/abs/1903.02531.

[5] Tingxiang Fan et al. "Fully Distributed Multi-Robot Collision Avoidance via Deep Reinforcement Learning for Safe and Efficient Navigation in Complex Scenarios". In: *CoRR* abs/1808.03841 (2018). arXiv: 1808.03841. URL: http://arxiv.org/abs/1808.03841.

[6] J. González et al. "Mobile robot localization based on Ultra-Wide-Band ranging: A particle filter approach". In: *Robotics and Autonomous Systems* 57.5 (2009), pp. 496–507. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2008.10.022. URL: http://www.sciencedirect.com/science/article/pii/S0921889008001747.

[7] Marcelo Segura et al. "Ultra Wide-Band Localization and SLAM: A Comparative Study for Mobile Robot Navigation". In: *Sensors (Basel, Switzerland)* 11 (Dec. 2011), pp. 2035–55. DOI: 10.3390/s110202035.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016. ISBN: 0262035618, 9780262035613.

[9] Maurizio.emilio. *Intelligenza artificiale, deep learning e machine learning: quali sono le differenze?* Feb. 2018. URL: https://www.innovationpost.it/2018/02/14/intelligenza-artificiale-deep-learning-e-machine-learning-quali-sono-le-differenze/.

[10] •. *Neuron To Neuron Related Keywords and Suggestions - Neuron To Neuron Long Tail Keywords*. 2019. URL: http://www.keywordbasket.com/bmV1cm9uIHRvIG5ldXJvbg/

[11] Daniel Graupe. *Principles of Artificial Neural Networks*. 3rd. WORLD SCIENTIFIC, 2013. DOI: 10.1142/8868. eprint: https://www.worldscientific.com/doi/pdf/10.1142/8868. URL: https://www.worldscientific.com/doi/abs/10.1142/8868.

[12] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[13] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

[14] David Silver et al. "Deterministic Policy Gradient Algorithms". In: *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*. ICML'14. Beijing, China: JMLR.org, 2014, pp. I-387–I-395. URL: http://dl.acm.org/citation.cfm?id=3044805.3044850.

[15] Timothy P. Lillicrap et al. "Continuous control with deep reinforcement learning." In: *ICLR*. Ed. by Yoshua Bengio and Yann LeCun. 2016. URL: http://dblp.uni-trier.de/db/conf/iclr/iclr2016.html#LillicrapHPHETS15.

[16] *Ultra-wideband RF System Engineering*. EuMA High Frequency Technologies Series. Cambridge University Press, 2013. DOI: 10.1017/CBO9781139058957.

[17] INC. ROBOTIS. *TURTLEBOT3*. 2019. URL: http://www.robotis.us/turtlebot-3/.

[18] INC. ROBOTIS. *TURTLEBOT3*. 2019. URL: http://emanual.robotis.com/docs/en/platform/turtlebot3/overview/.

[19] *809194 Robotis - TURTLEBOT3 Burger*. URL: https://www.robot-italy.com/it/robotis-turtlebot3-burger.html.

[20] *DYNAMIXEL All-in-one Smart Actuator*. URL: http://www.robotis.us/dynamixel/.

[21] J. J. Park and B. Kuipers. "A smooth control law for graceful motion of differential wheeled mobile robots in 2D environment". In: *2011 IEEE International Conference on Robotics and Automation*. May 2011, pp. 4896–4902. DOI: 10.1109/ICRA.2011.5980167.

[22] Tomas Berglund et al. "Planning Smooth and Obstacle-Avoiding B-Spline Paths for Autonomous Mining Vehicles". In: *Automation Science and Engineering, IEEE Transactions on* 7 (Feb. 2010), pp. 167–172. DOI: 10.1109/TASE.2009.2015886.

[23] Oussama Khatib Bruno Siciliano. *Springer Handbook of Robotics*. 2nd. Springer-Verlag Berlin Heidelberg, 2016.

[24] Monica Arias. *Autonomous Navigation by Reinforcement Learning*. Tech. rep. Universitat Jaume I, 2017.

[25] Amir Ramezani and Deokjin Lee. "An End-to-End Deep Reinforcement Learning-Based Intelligent Agent Capable of Autonomous Exploration in Unknown Environments". In: *Sensors* 18 (Oct. 2018), p. 3575. DOI: 10.3390/s18103575.

[26] Angelo Tartaglia. "Machine Learning Algorithms for Service Robotics Applications in Precision Agriculture". MA thesis. Politecnico di Torino, 2018.

[27] Vittorio Mazzia. "Use of Deep Learning for Automatic Low Cost Detection of Cracks in Tunnels". MA thesis. Politecnico di Torino, 2017.

[28] Csaba Szepesvari. *Algorithms for Reinforcement Learning*. Morgan and Claypool Publishers, 2010. ISBN: 1608454924, 9781608454921.

[29] Wikipedia contributors. *Lidar — Wikipedia, The Free Encyclopedia*. [Online; accessed 18-November-2019]. 2019. URL: https://en.wikipedia.org/w/index.php?title=Lidar&oldid=925009442.

[30] Giovanni Fantin. "UWB localization system for partially GPS-denied robotic applications". MA thesis. Politecnico di Torino, 2019.

Appendix A

Appendix A

Trajectories

In the following are presented all the trajectories in the different stages, both for humans and DRL agent.

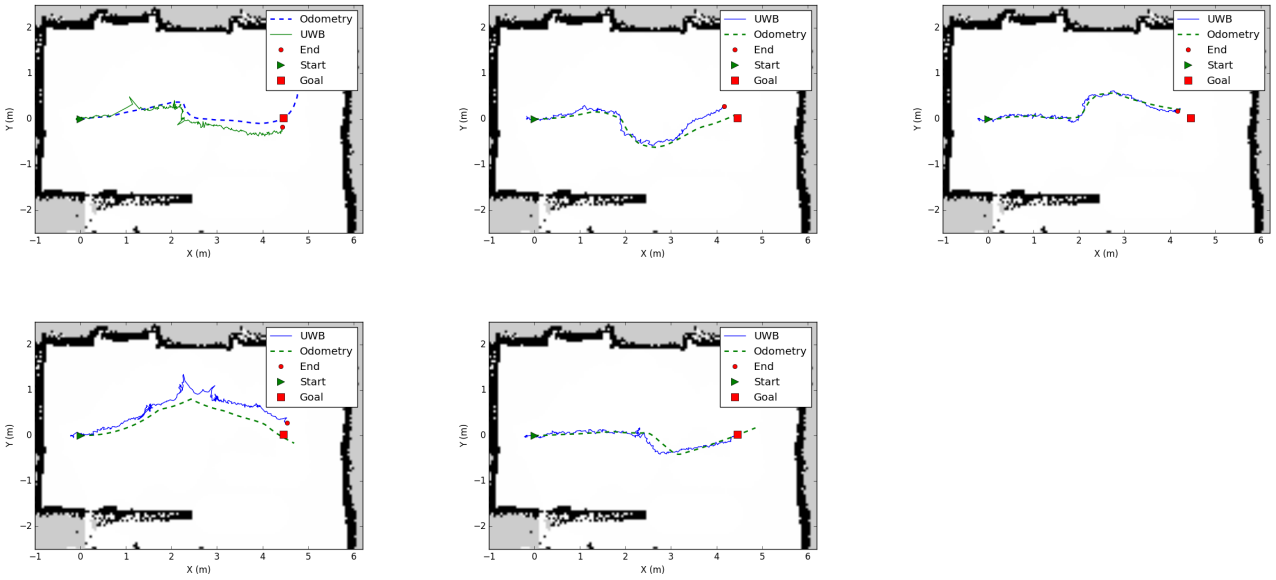


FIGURE A.1: Trajectories of agents in stage 1. In order: DRL, person 1, person 2, person 3, person 4

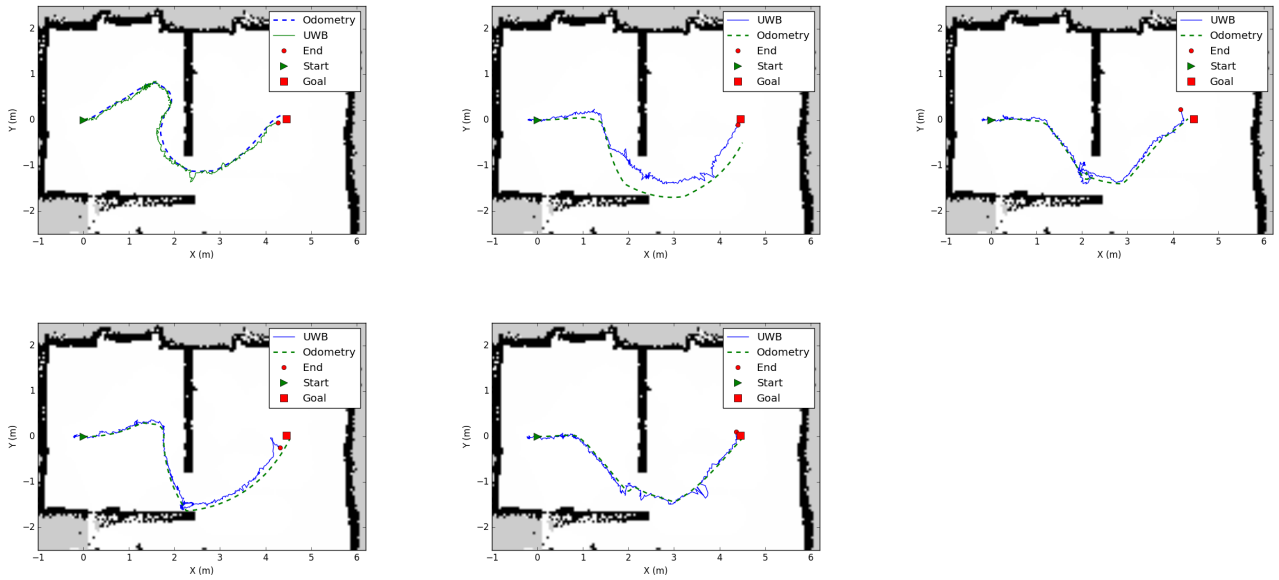


FIGURE A.2: Trajectories of agents in stage 2. In order: DRL, person 1, person 2, person 3, person 4

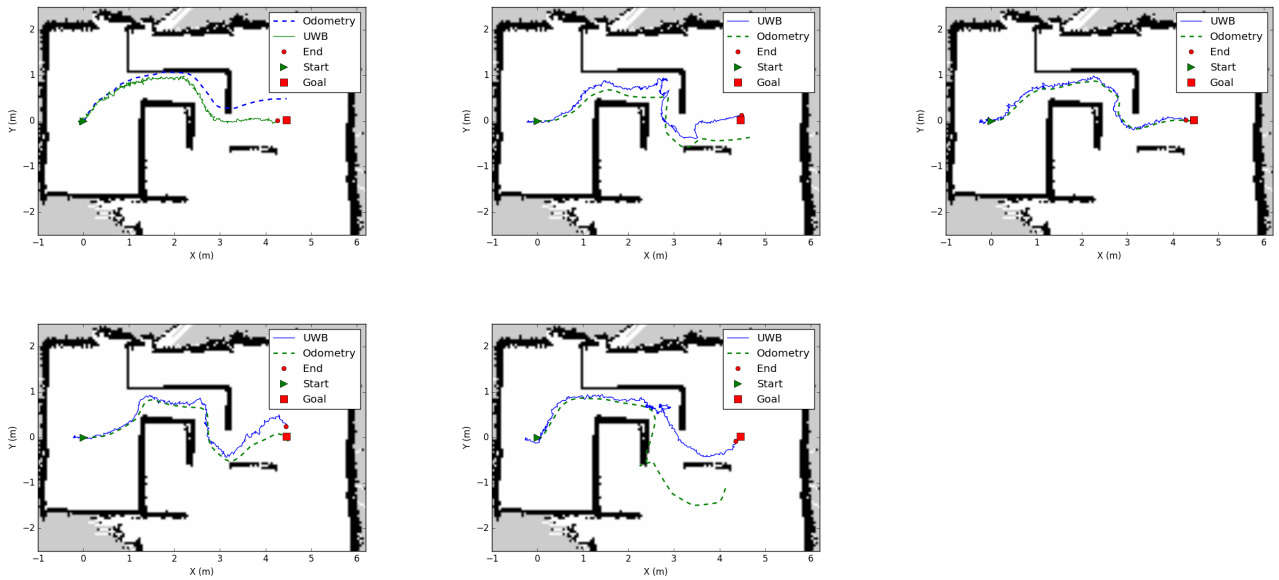


FIGURE A.3: Trajectories of agents in stage 4, task a. In order: DRL, person 1, person 2, person 3, person 4

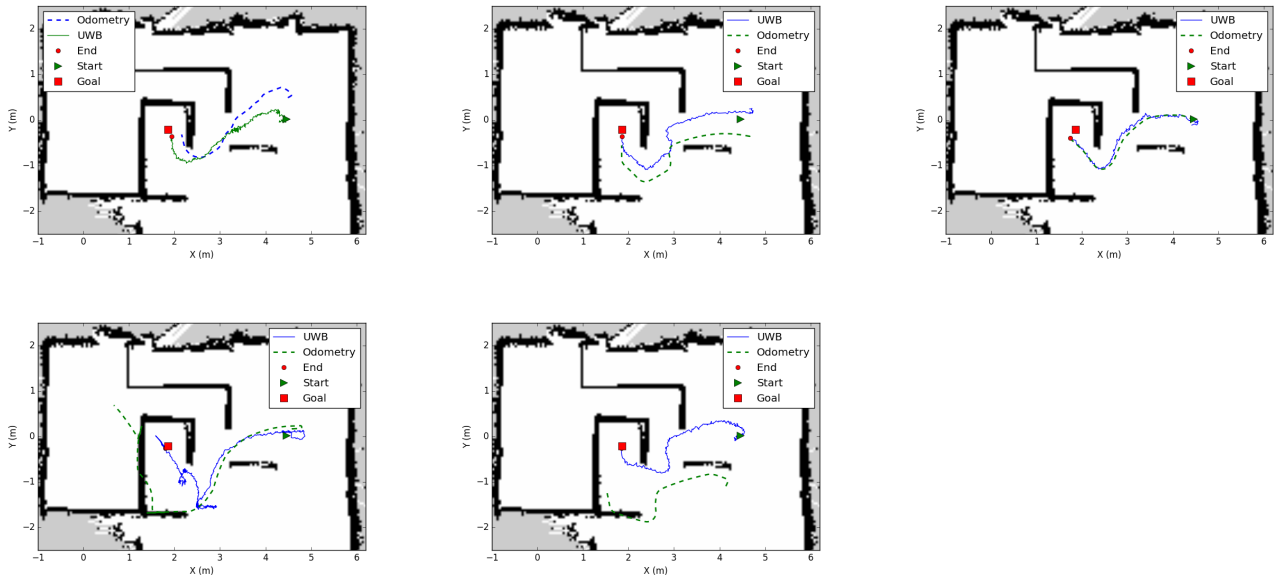


FIGURE A.4: Trajectories of agents in stage 4, task b. In order: DRL, person 1, person 2, person 3, person 4

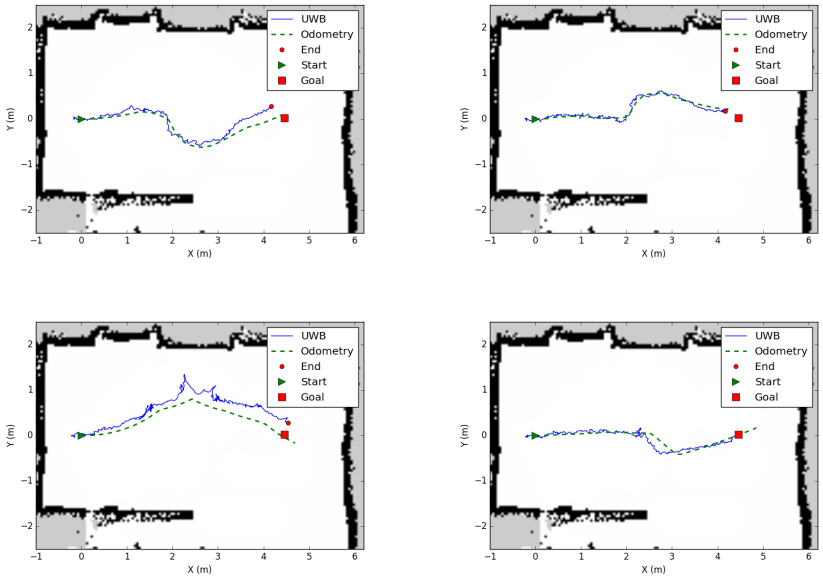


FIGURE A.5: Trajectories of agents in stage 4, task c. In order: person 1, person 2, person 3, person 4. DRL trajectory is not available.

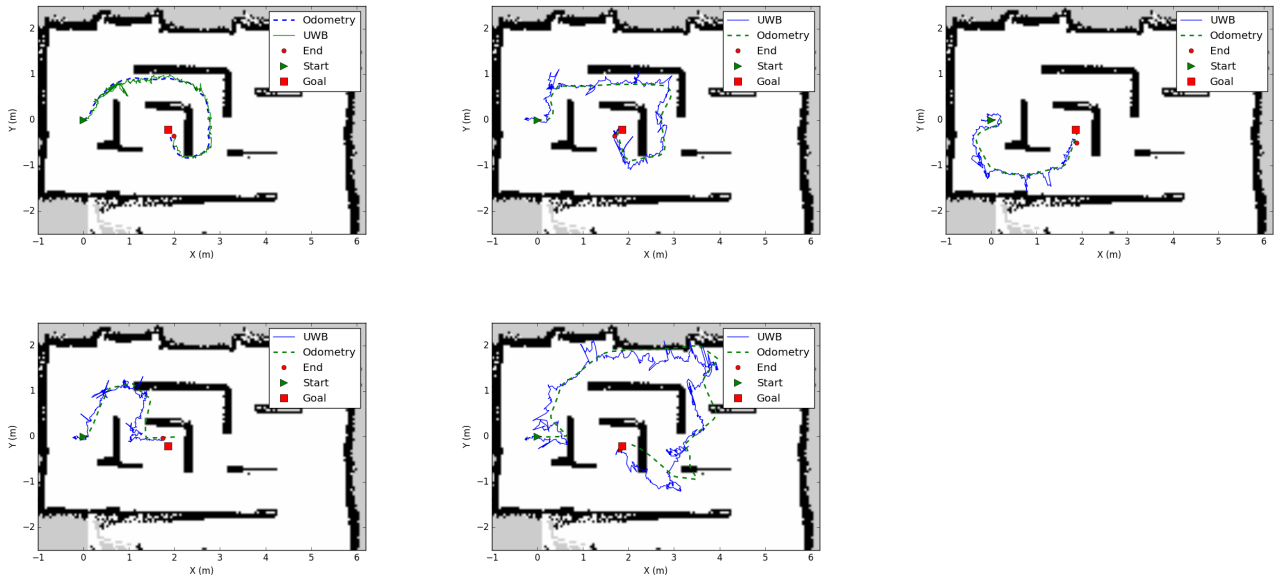


FIGURE A.6: Trajectories of agents in stage 5, task a. In order: DRL, person 1, person 2, person 3, person 4

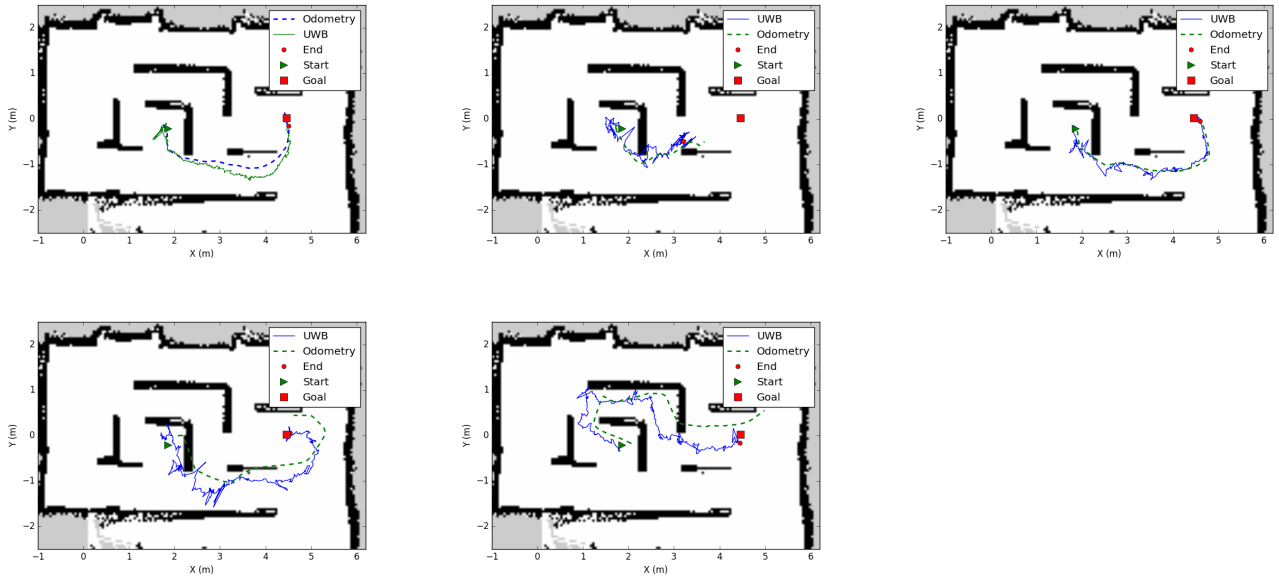


FIGURE A.7: Trajectories of agents in stage 5, task b. In order: DRL, person 1, person 2, person 3, person 4

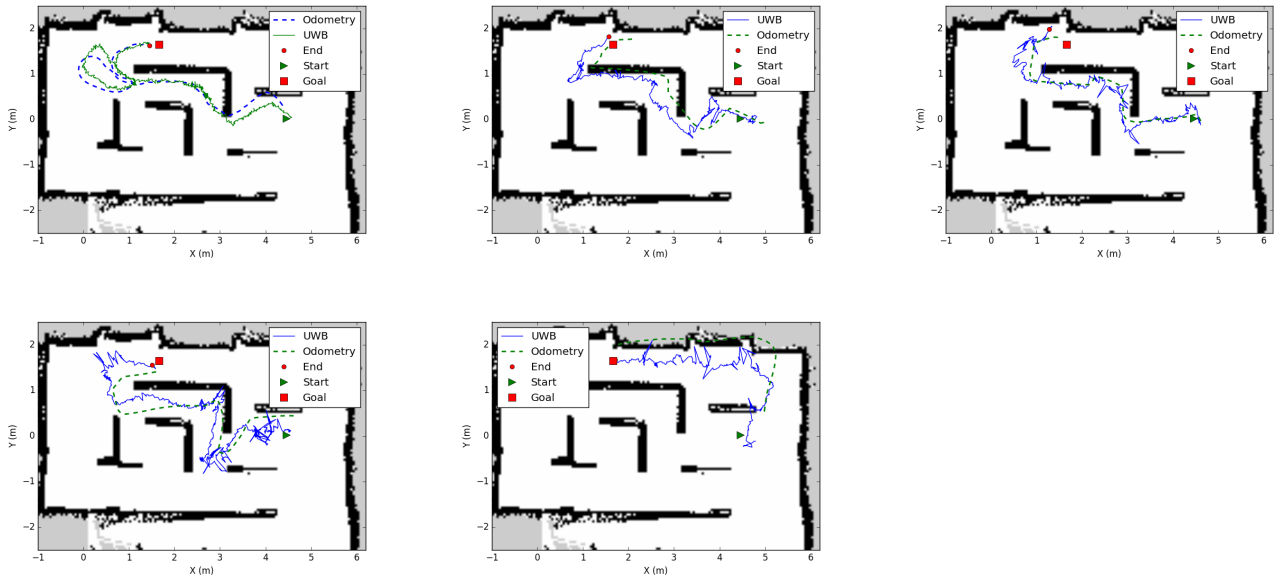


FIGURE A.8: Trajectories of agents in stage 5, task c. In order: DRL, person 1, person 2, person 3, person 4