POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

Learning Odometric Error in Mobile Robots with Machine Learning

Supervisors

Prof. Marcello CHIABERGE

Dott. Mauro MARTINI

Candidate

Alessandro NAVONE

October 2022

Summary

An accurate indoor localization for mobile robotic platforms is fundamental to accomplish autonomous navigation tasks. The most common source of odometric signal for wheeled Unmanned Ground Vehicles (UGV) is provided by the wheels' encoder sensors and an inertial measurement unit (IMU) if present. However, it is strongly sensible to slip conditions, accumulating error in the robot's positioning and becoming inaccurate after a short time since it is done integrating over the encoders' data. Visual odometry techniques can represent a precise alternative for an indoor environment with various features and suitable lighting conditions. However, when the environment offers few or repetitive visual patterns, it is necessary to consider an alternative for reliable UGV localization. Nonetheless, Ultra-WideBand (UWB) anchors recently emerged as a promising solution for indoor robot localization. UWB signals present high precision in Line of Sight (LOS) conditions. On the other hand, they are heavily subjected to a positive bias error in Non-Line of Sight conditions (NLOS) due to obstacles obstruction.

This work aims to study a Machine Learning-based solution to correct and improve robot localization techniques, relying only on raw sensor data. In particular, the study is focused on the correction of two cost-effective sensor signals: wheel odometry and Ultra-WideBand (UWB) anchors ranges. First, a dataset is collected running a Jackal UGV at the PIC4SeR (PoliTO Interdepartmental Center for Service Robotics) laboratory and recording the ROS topics published with the data measured by wheel encoders, IMU, UWB ranges of four antennas, and an Intel t265 visual odometry camera, used as ground truth and validated with the use of a Leica Absolute Tracker AT930. Different neural networks (NN) architectures are considered to correct the received odometry signals.
A classic feed-forward dense neural network (NN) for the UWB localization and a Long Short-Term Memory network (LSTM) for the encoder odometry. In both cases, the temporal sequence of previous N sensor data has been adopted to feed the NN models, demonstrating better performances than a single data sample. A grid search for hyper-parameters tuning has been carried out to select the best models on a validation data set.

Finally, for both the wheel odometry and UWB, an experimental test set is used to compare the selected best models with an Extended Kalman Filter (EKF) and an analogous NN trained with a single input instead of a temporal sequence of sensor data.

Table of Contents

List of Tables

List of Figures

Acronyms

AI
artificial intelligence

ANN
Artificial Neural Network

BP
Backpropagation

DL
Deep Learning

DNN
Dense Neural Network

EKF
Extended Kalman Filter

FFNN
Feedforward Neural Network

LSTM
Long Short-Term Memory

ML
Machine Learning

MLP
MultiLayer Perceptron

MAE

Mean Absolute Error

MSE

Mean Squared Error

NN

Neural Network

ReLU

Rectified Linear Unit

RMSE

Root Mean Squared Error

RNN

Recurrent Neural Network

UAV

Unmanned Aerial Vehicle

UGV

Unmanned Ground Vehicle

UWB

Ultra-WideBand

Chapter 1

Introduction

1.1 Objective of the thesis

Autonomous navigation is one of the most challenging skills to achieve in mobile robotics, and its success depends on an accurate localization system; in particular, in indoor localization, high precision and robustness are required. Nonetheless, if the odometric signal is obtained by wheel encoders or inertial measurement units only, it lacks robustness and accuracy due to the accumulation of the error, which tends to explode after a few times making the position data unreliable. Literature offers many approaches to tackle this problem and mitigate the error, mainly based on Extended Kalman Filters; however, due to the growth in popularity of Deep Learning, many Neural Network-based solutions have been proposed. Moreover, during the development of this work, we decided to try to study simultaneously a localization system based on Ultra WideBand anchors, which is emerging as a promising solution for indoor localization. UWB anchors present a high accuracy in conditions of Line of Sight while suffering a heavy positive bias in case of Non-Line of Sight conditions.

The aim of this thesis consists in mitigating the error of mobile robot localization in the aforementioned situations through a Deep Neural Network-based solution. In particular, the use of Long Short-Term Memory Neural Network is explored with sequence-shaped inputs. The data used to train and test the proposed models are collected in an indoor environment. In particular, a Clearpath Jackal mobile robot has been teleoperated in an indoor space to collect the data of the sensors. In particular, an Intel Realsense tracking camera t265 and a Leica Laser Tracker AT403 have been used as a ground truth reference, meanwhile, a DecaWave TREK1000 kit has been used to collect UWB data. A particular focus is made on the research and tuning of the hyperparameters of the Neural Network model. Finally, the results will be compared with the position signals obtained through Extended Kalman

Filters.

This project was born at the PIC4SeR (PoliTo Interdipartimental Center for Service Robotics) in a wider context that focuses on service robotics for the development of a highly innovative solution for various fields such as precision agriculture, smart cities, wellbeing, cultural heritage and space applications. In this context, an accurate localization system concurs in achieving autonomous navigation tasks, which represents a competitive advantage in all these application fields.

1.2 Organization of the the work

This thesis is subdivided into the following chapters, which are briefly described in this section.

Chapter 1 introduces the context and the goal of the thesis, summarizing also the adopted methodology.

Chapter 2 introduces the localization problem, providing a model for odometric position and illustrating the main source of error in wheel odometry. Then the UWB technology is presented by defining regulation, the adapted terminology, the main ranging methods and the main sources of error.

Chapter 3 offers a broad panoramics of the state of the art of both localization techniques. In the first section, an overview of the main approaches to localization is given, and then, in the following section, some examples of Machine Learning and the respective methodology found in the literature are illustrated.

Chapter 4 introduces the main elements of machine learning, starting from a historic perspective; subsequently, the main bricks of Deep Learning are explained, such as architectures and training algorithms. The last part of the chapter focuses on recurrent neural networks and specifically on LSTM structure.

Chapter 5 describes the robot platform, the sensors used and the software used to develop this project.

Chapter 6 is devoted to illustrating in detail the development process of this work. First, the dataset creation and its features are described. Later the grid search results are analyzed, and finally, the results are presented.

Chapter 7 ends the thesis, with the conclusions of the work and possible future developments.

Chapter 2

Introduction to the localization problem

This chapter aims at introducing the localization problem, that represents the core of this work. In the first section, a general overview and context about the localization are given, with a general mathematical formalism to represent the odometry of a mobile robot in a plane, its update during the time and its error. Then, the main causes of wheel odometry errors are introduced. The final section of this chapter introduces the reader to the UWB technology for localization, the main ranging techniques and the primary sources of error.

2.1 The localization problem

Navigation is a fundamental task for mobile robot platforms and it consists of four main phases. [1]

- *Perception*: the extraction of meaningful data from sensors
- *Localization*: the determination of the position in the environment
- *Cognition*: the decision-making process to reach a goal
- *Motion control*: the actuation of the correct movement to cover a trajectory

The rest of this thesis will focus on the localization problem.

It is possible to classify the localization problems based on the available knowledge of the initial position and face each case with different methods.

- *Position tracking:* the initial position is known and the robot is tracked at every time instance during its navigation in the environment. The robot position is continuously updated using the previous position by using odometry and sensors. Therefore, uncertainty should be kept bounded, otherwise, the robot may not be localized. Furthermore, in this case, the initial position belief is a normal distribution

- *Global localization*: the initial position is not known and the robot has to locate itself within its environment

- *Kidnapped robot problem*: the robot has to relocate itself in its environment and uses the match between the current data of the sensors and is knowledge about its position. An autonomous robot should be able to monitor its pose and be able to relocate itself when it recognises it has been kidnapped.

To confront these three problems, many strategies have been taken into account: *probabilistic localization strategies*, *evolutionary strategies*, *automatic map-based localization* and *RFID* approaches. [2]

As a mobile robot navigates around an environment, it keeps track of its movement through odometry, but its uncertainty makes the self-localization of the robot not reliable. To address this issue, the localization process should take into account the map of the environment, preventing the error to grow without limits. Therefore, in addition to wheel odometry, exteroceptive sensors such as lasers, cameras and ultrasonic sensors, can collect data about the environment. It is not possible to measure the accurate position directly, but it can only be estimated in two phases. [1]

- *Prediction update phase*: during this phase proprioceptive sensors such as wheel encoders and IMU are used to estimate its position, given a known initial position x_0. During the navigation, the error accumulates and uncertainty grows.

- *Perception update phase*: the robot uses exteroceptive sensors to update its position and correct, if necessary, the position estimated in the previous phase, reducing the uncertainty.[2]

2.1.1 A model for odometric position

The pose of a mobile robot can be describet with three coordinates.

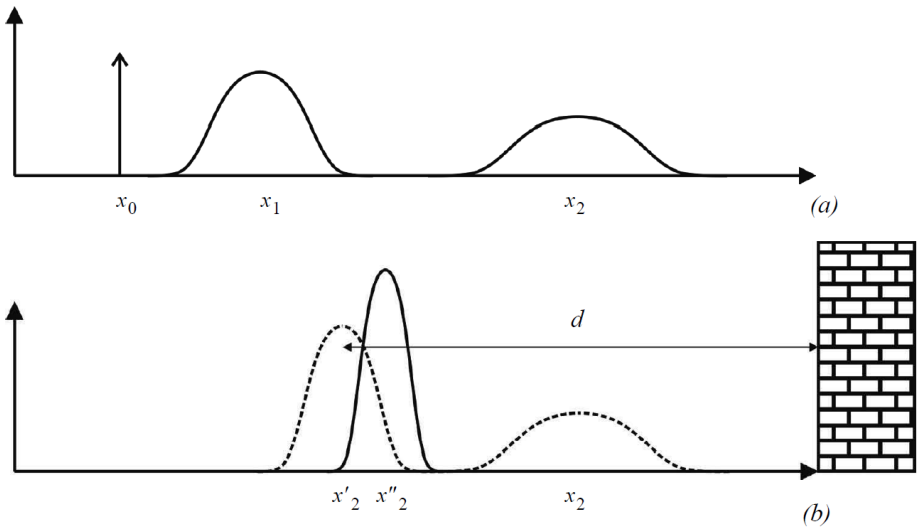


Figure 2.1: (a) *Prediction phase*: the position at the initial time instant is known, so the probability density function is a *Dirac delta*. When the motion starts, the uncertainty starts to accumulate and the p.d.f. changes. (b) *Perception phase*: exteroceptive sensors are used to measure the position with respect to the environment, shrinking the uncertainty. [1]

$$\mathbf{q} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \tag{2.1}$$

In the case of a differential drive robot, it is possible to define the incremental travel distances during a fixed sampling interval Δt.

$$\Delta x = \Delta s \cdot cos\left(\theta + \frac{\Delta\theta}{2}\right) \tag{2.2}$$

$$\Delta y = \Delta s \cdot sin\left(\theta + \frac{\Delta\theta}{2}\right) \tag{2.3}$$

$$\Delta\theta = \frac{\Delta s_r - \Delta s_l}{d} \tag{2.4}$$

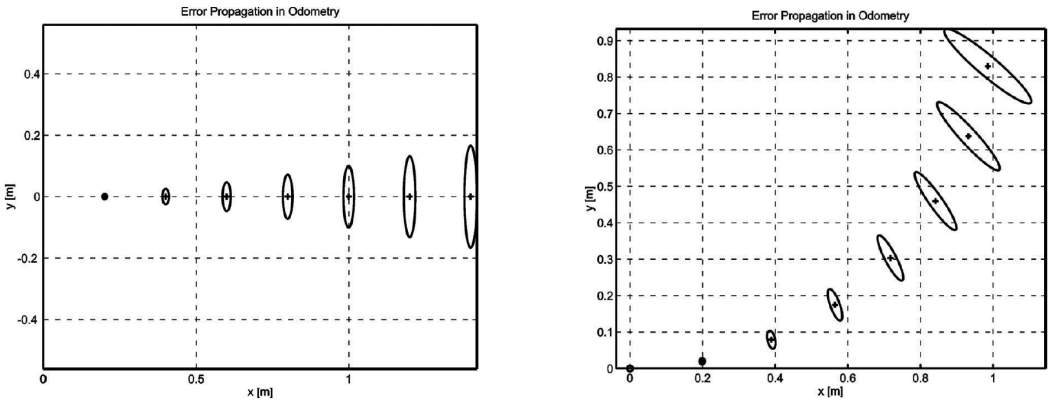
Where Δs_r and Δs_l are the distances accomplished by the left and right wheel respectively and d is the distance between the wheels.

So now it is possible to write the equation for the update of the position. [1]

$$p' = \begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} + \begin{bmatrix} \Delta s \cdot cos\left(\theta + \frac{\Delta\theta}{2}\right) \\ \Delta s \cdot sin\left(\theta + \frac{\Delta\theta}{2}\right) \\ \Delta\theta \end{bmatrix} \tag{2.5}$$

2.2 Error sources in wheel odometry

The error of wheel odometry, if not corrected, rapidly tends to accumulate and explode. It is possible to compute its value by defining an "error ellipse", which describes the region of the uncertainty of the actual position as in the figures 2.2a and 2.2b. It is shown how its dimensions grow along the trajectory until the error makes the position value useless and unreliable. It can be seen that in a straight motion along the x-axis, the uncertainty grows faster along the y-axis: this happens because of the uncertainty of the orientation θ.



(a) Uncertainty ellipse in straight motion: the uncertainty grows along the y axis due to the integration of robot orientation.

(b) Uncertainty ellipse in curve motion: the uncertainty mainly grows along the direction perpendicular to the motion but the main axis of the ellipse is not perpendicular to it.

Figure 2.2: Uncertainty ellipses. [1]

If we focus on differential-drive robots, it is possible to divide the sources of error into two main categories: *systematic errors* and *non-systematic errors*. The main causes of systematic errors, according to Borenstein and Feng, [3] are:

- different mean radius between the two wheels

- non-constant wheel radius
- non-aligned wheels
- encoder resolution
- encoder sampling rate
- uncertainty of effective wheelbase

The main causes of non-systematic errors are:

- uneven surfaces
- unexpected objects on the surface
- wheel-slippage due to slippery surface, acceleration, skidding, external forces and wheels not touching the floor

Systematic errors can cause the worst effect if not corrected, due to their integration. Nonetheless, it is possible to measure and correct them easily.

2.3 Ultra-Wideband for mobile robot localization

2.3.1 Definition of UWB signals and regulations

According to the IEEE 802-15a standard, an ultra-wideband signal satisfies one of the following two conditions: [4]

- its *simultaneous bandwidth* B is equal to or greater than 500 MHz.

$$B \geq 500MHz \tag{2.6}$$

- its *fractional bandwidth* f_r, defined as in formula (2.7), is larger than 20 %. f_u and f_l are the upper and lower frequencies where the power spectral density is $10dB$ below its maximum, and f_c is the central frequency.

$$B_r = \frac{f_u - f_l}{f_c} \tag{2.7}$$

Each country has different regulations concerning the spectrum mask of UWB signals which define the application of UWB technology, emissions level (*Power Spectral Density*), interference mitigation and allocated frequency ranges. The first country to release UWB regulation was the United States, in 2002, through the *Federal Communication Commission* (FCC), followed by European Union in 2006

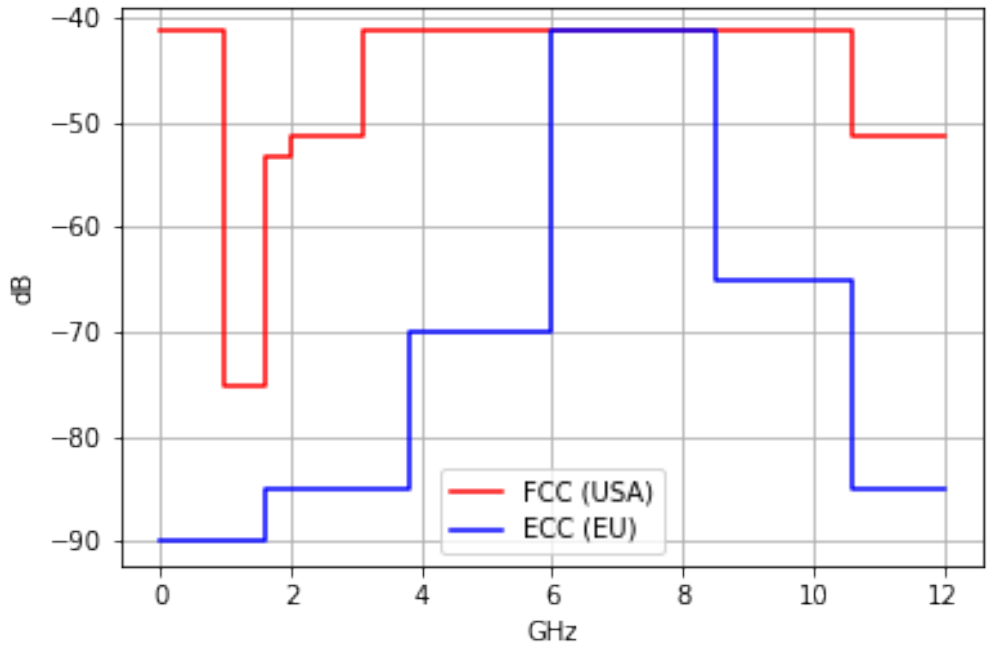


Figure 2.3: UWB spectral masks in USA and EU [4]

through *European Communication Commission* (ECC). American and European regulations are summarized in figure 2.3.

The maximum emission level is very low in all cases ($-41.3dBmMHz$), but the ultra-wideband allows to realize high data rates ($> 100Mbit/s$); thus, to fulfil regulations, UWB commercial applications are limited to short-range applications only. Two main approaches are used to exploit the spectrum characteristics:

- *Impulse Radio* (IR) UWB is based on ultra-short pulses radiated directly from the antenna. It is not efficient in the exploitation of the mask and is subject to the effects of the noise but can be achieved with simple analog components.

- *Orthogonal Frequency Division Multiplexing* (OFDM), where the UWB spectrum is divided into a set of broadband OFDM channels. In this case, the orthogonality of subcarriers avoids cross-talks, which leads to more efficient exploitation of the band and more robust transmission in the case of noise, but it is more complex to achieve in terms of signal processing.

2.3.2 Terminology

Before explaining the different techniques used to estimate the position of a mobile robot using UWB signals, it is necessary to introduce a typical experimental scenario with specific terminology. [5]

- An *anchor* is a fixed UWB transmitter/receiver which is used as the reference point. Usually, several anchors are used to obtaining a 2D or 3D position.
- The *tag* is the moving UWB transmitter/receiver attached, in this case, to the mobile robot platform to locate. It communicates with the anchors.
- The *direct path* is the straight line from a tag to an anchor
- An *obstacle* obstructs is an object that may interrupt the direct path between an anchor and the tag.
- *Multipath component*s are the non directs signal paths originating from reflections in the environment.

2.3.3 Ranging methods

Several techniques can be used to estimate the distance between two UWB sensors: the ranging estimation is then used to estimate the position of the tag in an environment. [4]

Received Signal Strenght (RSS)

As the signal travels in space, going further from its source, it becomes weaker. Knowing the initial transmitted power, it is possible to estimate roughly the distance between the transmitter and the receiver. Despite the ease of implementation, this method suffers from poor accuracy, especially for an indoor environment where it is not possible to assume free space propagation. The first approach to mitigate this problem and exploit RSS is the so-called *fingerprint*, which consists in creating a map of the power of the received signals in the environment. However, this approach's main flow is a poor generalization, since a new map should be done for every considered space. Another approach that can be used is *trilateration*, considering at least the received power from three anchors, estimating their distance from the tag and obtaining the position.

Time of Arrival (TOA)

The *time of arrival* ranging technique consists in measuring the time employed by the signal to travel from the transmitter to the receiver, also called *Time of*

Flight (TOF), and estimating the distance between them. Time of Flight can be computed since the received messages contain the sending time: this technique is highly error-prone in case of a non-perfect synchronization of the transmitter and the receiver. *Two Way Ranging* (TWR) protocol can solve the problem of device synchronization, measuring the time only on a single device: a signal is sent from the tag to the anchor, which after a known time of reply sends the message back. Given the time of a round T_{round} and the time of replay T_{reply} it is possible to calculate the time of flight T_{tof}.

$$T_{tof} = \frac{1}{2}(t_{round} - t_{reply}) \tag{2.8}$$

However, this method is sensitive to the bandwidth and Non Line of Sight (NLOS) conditions. As the RSS, computing the distance between the anchors and the tag allows for the application of a trilateration algorithm to locate the tag in the environment.

Time Difference of Arrival (TDOA)

Time difference of arrival (TDOA) measurement, contrary to the TOA case, requires only the synchronization of the anchors themselves, which are placed in known positions. Measuring the difference in the arrival time of two signals allows us to calculate a hyperbola, a set of points indicating a constant distance between two anchors, where the tag can be located. Since the hyperbola is a 3D surface, it is necessary to place at least three anchor nodes to intersect at least two hyperboloids, to find the point where the tag is positioned. The TDOA approach, as TOA, heavily suffers from Non Line of Sight conditions.

Angle of Arrival (AOA)

The measurement of the *angle of arrival* (AOA) provides the direction of the incoming signal, that is the angle between two nodes. This means that if the tag measures the angles from two anchors, it is enough to estimate the tag's position in a 2D environment, without the need for synchronization. . On the other hand, the hardware needed to carry out angle measurements is more complex, such as antennas array.

2.3.4 Main sources of error

It is possible to identify the main sources of error in UWB ranging measurements, focusing mainly on the Non-Line of Sight problem [5].

- *Multipath propagation*: in a multipath environment the reflection of the signal may lead to errors since the TOA estimation is based on the time shift of a template signal. Fortunately, this error is mitigated by the large band of UWB without the need for a complex algorithm.
- *Multiple access interference*: it happens when more than one nodes are active and their signals interfere with and degrade the ranging performance. This error can be mitigated by a training code that sets up the transmitters and the receiver.
- *High-time resolution*: the practical implementations of UWB sampling above the Nyquist frequency are impractical since it requires a sampling frequency around the tens of GHz. Moreover, clocks drift and jitter affect the accuracy.
- *Non-Line of Sight*: if the direct path is obstructed by obstacles, which could be static such as a wall or a piece of furniture, or dynamic, such as a person. In this case, two situations are possible: soft NLOS, when the obstacle weakens the strength of the signal which is received with a lower amplitude, and hard NLOS, when the signal does not reach directly the receiver, so the reflections are detected.

Chapter 3

State of the art

The localization of a mobile robot is a fundamental step in the autonomous navigation process, and in literature, many studies and approaches can be found. The first section of this chapter illustrates to the reader the main techniques employed nowadays, including the most used sensors and their advantages and drawbacks. Since this work focuses on the exploitation of Deep Learning for localization, the second section of this chapter focuses on the affine works to the subject of this thesis that can be found in the literature.

3.1 Mobile robot localization: sensors and techniques

Two main classes of localization approaches can be identified: probabilistic approaches and autonomous map building. In the first case, Markov localization and Kalman Filter (KF) are the main techniques that can be found in the literature and can be applied to a wide variety of sensor data.

3.1.1 Probabilistic approaches

Markov localization

At the beginning, the robot can locate itself starting from an unknown position, and then track multiple possible positions, so it is possible to recover from ambiguous situations. It is necessary to represent the state space in a discrete way, such as a topological graph or a grid, in order to update the probability of possible positions. The available information about the previous position and the odometry input is used during the update of the predicted localization. Therefore, considering the previously estimated position $\overline{S_{best}}(x_{t-1})$ and the proprioceptive data, which are

used as a control input u, the current state $\overline{S_{best}}(x_t)$ is evaluated. The following equations cover respectively the discrete (3.1) and the continuous (3.1) case.

$$\overline{s_{best}} = \sum_{x_t - 1} p(x_t)|u_t, x_{t-1})S_{best}(x_{t-1}) \tag{3.1}$$

$$\overline{s_{best}} = \int p(x_t)|u_t, x_{t-1})S_{best}(x_{t-1})dx_{t-1} \tag{3.2}$$

In case the measurement is done using exteroceptive sensors data z_t combined with the earlier estimated position$\overline{S_{best}}(x_t)$, the Bayes rule is used to compute the robot's new state $S_{best}(x_t)$.

$$S_{best} = \eta p(z_t|x_t, M)\overline{S_{best}}(x_t) \tag{3.3}$$

Where x_t is the robot position and $p(z_t|x_t, M)$ is the probabilistic model, which represents the probability of the observation of z_t given the robot poses x_t and the map M. Moreover, the noise-free measurement function h depends on X_T and M. Finally, a noise term is added to the measurement function h, to derive the probabilistic measurement model, such that the probabilistic distribution $p(z_t|x_t, M)$ has a peak at $h(x_t, M)$. Considering the noise to be a gaussian function, the following function is derived.

$$p(z_t|x_t, M) = N(h(x_t, M), R_t) \tag{3.4}$$

Where N is a multivariate normal distribution with mean $h(x_t, M)$ and covariance matrix R_t. [2]

Kalman Filter

The *Kalman Filter* (KF) approach is an efficient and precise method to solve the tracking problem: the position is tracked from an initial *known position* with an optimal sensor fusion approach. Generally, this method can be used in a continuous state representation, with some exceptions in case the uncertainty is too large: in this case, the position is lost and cannot be recovered anymore. Kalman filter can be considered a particular case of Markov localization, where *Gaussians* are used to represent the robot position assumption S_{best}, the motion model and the measurement model. In the first phase, namely the *prediction update*, a motion model with a Gaussian error is applied to the measured encoder data. The second phase, namely the *perception update*, consists of the following four steps:

- *Observation step*: different features are extracted from sensor data.
- *Measurement/Perception step*: a measurement prediction is produced, containing the expected features to be observed from the predicted position during the prediction step.

- *Matching step*: the best match between observed features and expected features evaluated during the prediction step is computed.
- *Estimation step*: KF fuses the matching information to update the robot's belief state during the estimation step.

The normal probability density function (PDF) for a Gaussian distribution is defined as follows.

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{3.5}$$

Where x is a scalar random value, μ is the mean of the scalar random value x and σ^2 is its variance.
In case x is a multidimensional vector, with k dimensions, it is possible to define a multivariate normal distribution.

$$p(x) = \frac{1}{\left|\sqrt{\Sigma}\right| \sqrt[k]{2\pi}} exp\left(-\frac{(x-\mu)^T}{\Sigma^{-1}(x-\mu}\right) \tag{3.6}$$

Where μ is the vector of the mean values and Σ is the covariance matrix, which is symmetric and positive-semidefinite.

Even though Kalman Filter is an optimal solution for the tracking problem, it does not solve the kidnapped robot problem and global localization problem. Many robotics applications require a non-linear system: in this case, the *Extended Kalman Filter* (EKF) is used, which linearizes around the estimate of the current mean and covariance. This can be considered the standard approach for localization problems.

3.1.2 Simultaneous Localization and Mapping (SLAM)

The previous probabilistic approach required an environment map, which is, in most cases, hand-made: the robot uses environment features, that must be included in the map, to locate itself. Having a previously designed map can be challenging for complex, large or dynamically changing environments. Thus, *automatic map building* is a solution to this problem: the robot starts the navigation from a random point and explores the environment through its sensors gaining knowledge of the environment and building the map. [1]

The main issue consists in creating the map and modifying it automatically, which in robotics is called *simultaneous localization and mapping* (SLAM). The environment map is built by obtaining accurate information about the robot's path, which estimation requires a precise map: the whole process is done by collecting

data from proprioceptive and exteroceptive sensors. Generally, the robot path is obtained from the odometry data and environment features, collected by sensors such as cameras, ultrasonic sensors or LiDaRs. Usually, the formulation of the SLAM problem is probabilistic and the map is represented as a probability distribution: the objective is to estimate the robot's position and the map from sensors' observations during the time.

We'll denote the robot path as X_T, its relative motion as U_T and the environment map as M.

$$X_T = \{x_i | 0 \leq i \leq T\} \tag{3.7}$$

$$U_T = \{u_i | 0 \leq i \leq T\} \tag{3.8}$$

$$M = \{m_i | 0 \leq i \leq n-1\} \tag{3.9}$$

The initial position x_0 is known, and the motion during a time step, namely between $t-1$ and t, is indicated as u_t. The sequence of observations in the robot's reference frame is defined as:

$$Z_T = \{z_i | 0 \leq i \leq T\} \tag{3.10}$$

Thus, the SLAM process consists of obtaining the map model M and the robot trajectory X_T given the odometry U_T and the data observations Z_T. It is possible to face the problem in two ways:

- *full SLAM*: the full path is updated, consisting of the estimation of the joint posterior probability of X_T and M, namely $p(X_T, M | Z_T, U_T)$.
- Online SLAM: at every timestep, the robot current position x_y and the map are updated, namely $p(x_t, M | Z_T, U_T)$.

The SLAM process can be addressed using *extended Kalman filter* (EKF) or *unscented Kalman filter* (UKF) approaches. [2]

3.1.3 Sensor for indoor localization

The traditional localization techniques mainly rely on GPS signal, which works well for open outdoor environments, but is not suitable for indoor spaces, where the signal is reflected or impeded by walls and its accuracy is too large. In robotics literature, many studies about self-contained odometry methods and simultaneous localization and mapping (SLAM) can be found: these techniques can be implemented relying

on data obtained from onboard sensors. In this case, odometry uses local sensors to estimate the pose of a mobile robot starting from a known position. The following part of the section offers a broad panoramic view of the main types of sensors and sensor fusion used to estimate the odometry of mobile robots, all their features are summed up in the table 3.1. [6]

	real time	power	accuracy	robustness	dimensions
GPS	soft	low-power	semi-accurate	high	2D
WO	hard	low-power	non-accurate	low	2D
IO	hard	low-power	non-accurate	high	3D
LO	hard	high-power	accurate	medium	3D
VO	firm	high-power	accurate	low	3D
FB VIO	firm	high-power	accurate	medium	3D
OB VIO	soft	high-power	accurate	medium	3D
LC VIO	firm	high-power	non-accurate	low	3D
TC VIO	soft	high-power	accurate	medium	3D

Table 3.1: Comparison of common odometry techniques [6]

Wheel odometry (WO)

It is one of the simplest forms of odometry, applied to many differential robots, such as two or four-wheeled platforms. The number of revolutions made by each wheel is measured by wheel encoders and then integrated through a dynamic model to obtain the current position. This method is subjected to drift and slippage, and it is not suitable for long-term reliable localization.

Inertial odometry (IO)

It uses the measurement from the IMU sensor to determine the position and the orientation of the robot in a 3D space: the inertial sensor consists of a 3-axis accelerometer and a 3-axis gyroscope. As the wheel encoder odometry, this method is subjected to drifting during the time, due to the IMU inaccuracy, and is not suitable for long-term localization. To contain this problem, often a double-integration based on an Extended Kalman Filter (EKF) is used.

Laser odometry (LO)

Laser odometry is based on tracking the laser speckle patterns reflecting on the surrounding environment on a 2D observation plane. A 3D space is reconstructed by superposing several 2D images. This method is not addressed in this work.

Visual odometry (VO)

It estimates the pose of the mobile robot by analyzing the changes in the frames of the camera over time. The main features of an image are extracted using an image feature detector. Three main techniques are used to evaluate visual odometry:

- *Direct approach*, where raw measurements (i. e. pixels) are used to estimate the position of the robot, detecting the change in the different frames. This solution requires many equations.
- *Feature-based approach*, where only the movement in the points of interest, such as corners or edges, are considered. The pose of the robot is estimated by minimizing the geometric error to calculate the transformation matrix: this allows for considering only a few features.
- *Hybrid approaches*: the feature-based approach can lack robustness in low-textures environments because they manage to extract only a few features. Hybrid approaches tackle this problem by merging the two previous methods.

Visual-laser odometry

In this case, LiDaR and visual odometry are fused to avoid the problem generated by the limitation of the laser, such as motion distortion, and the camera, such as low-texture environments.

Visual-inertial odometry (VIO)

Integrating visual and inertial odometry helps to overcome the limitation of both approaches: IMU sensor is not affected by the surrounding conditions while camera data do not deteriorate with time. The resulting visual-inertial odometry (VIO) is characterized by great robustness and accuracy. VIO can be categorized as filter-based and optimization-based, which depends on how the data are fused, and in loosely-coupled and tightly coupled.

- *Loosely-coupled approach* (LC): the position and the orientation of the robot are obtained by merging the estimation of the two isolated systems. Generally, the data fusion takes place through a Kalman Filter.
- *Semi-tightly coupled approach*: it consists in fusing the estimation provided by visual odometry with the row data from the IMU to achieve accuracy.
- *Tightly-coupled approach* (TC): the main tracked features extracted from images are fused with raw measurements of the IMU. This leads to better performances than other coupling methods.

- *Filter-based approach* (FB): this approach was one of the first to tackle VIO and SLAM problems. It consists of two parts: the prediction step and the update step. It is possible to consider them as a maximum a posteriori (MAP) technique, where the proprioceptive measures (IMU) are used as prior distribution of the position and the exteroceptive sensors contribute to constructing the likelihood distribution. Three types of filters are used: Extended Kalman Filter (EKF), Multi-State Constraint Kalman Filter (MSCKF) and Unscented Kalman Filter (UKF).
- *Optimization-based approach* (OB): it consists in estimating the position by optimizing main features extracted from camera images and inertial measurements jointly. This technique is based on iteratively minimizing the least square function.

3.2 Machine learning for robot localization

3.2.1 Machine learning for wheel odometry

Estimating the Odometry Error of a Mobile Robot by Neural Networks (H. Zu, J. J. Collins, 2009) [7]

This work presents different approaches to differential mobile robot localization which can correct both systematic and non-systematic errors through a neural network. The robot's actual trajectory, used as ground truth, is captured via a laser scanner which measures the position and motion of a marker, which consists of a white tube placed on top of the robot. The two-layer perceptron is fed with the data provided by a generic third-party localization module. The dataset consists of samples of eight (x, y) points; 80 % of the data is used to train the network and the remaining 20 % to test it using cross-validation. The testing phase has been carried out by making the robot follow different trajectories (a straight line, a rectangle and a curve): the evaluation is based on computing the Mean Absolute Percentage Error (MAPE) on the samples of eight points and then compared to the UMBmark proposed by Borenstein and Feng [3].

Learning Uncertainties in Wheel Odometry for Vehicular Localisation in GNSS Deprived Environments (U. Onyekpe, V. Palade, S. Kanarachos, S. Christopoulos, 2020) [8]

This work does not address mobile robots but vehicle localization, in particular the position update in case of GNSS outages using wheel encoders instead of the Inertial Navigation System (INS). An LSTM network has been used, fed with wheel speed and GPS data taken from different datasets recorded in different environments and

driving conditions, with a time window of 11 s, with a sampling frequency of 1 Hz. The performance of the proposed method has been tested on a part of the dataset not used for training and comparing the cumulative Root Squared Error (CRSE) with the traditional INS method in different situations.

Learning Wheel Odometry and IMU Errors for Localization (M. Brossard, S. Bonnabel, 2019)[9]

This work uses a differential robot dataset [10] recorded in an outdoor environment to train a Gaussian Process (GP) to learn the residual between ground truth data and a position estimation calculated using a deterministic model based on wheel encoders or IMU. A machine learning approach has been used to ensure scalability. Finally, an Extended Kalman Filter that uses the corrected model with automatic differentiation for the computation of the Jacobians is used to compute the robot's position. A part of the dataset is left for the testing phase: when the proposed approach is compared to the physical model analyzing the mean Absolute Trajectory Error (m-ATE) and the cumulative Absolute Trajectory Error (c-ATE) on position and orientation.

Drift Compensation of a Holonomic Mobile Robot Using Recurrent Neural Networks(K. O. Canbek, H. Yalcin, E. A. Baran, 2022)[11]

This work focuses on correcting the odometry error of a mobile robot platform with four mechanum wheels through a GRU neural network, using the raw data from wheel encoders and accelerometers and the ground truth is provided via an optoelectronic motion tracking device. The sensor data rate is equal to 1000 Hz and the yaw motion is not provided. To identify the best structure of the network a grid search has been done and the best model is then tested over some specific trajectories. A Kalman Filter-based solution has been used as a benchmark comparing the root mean squared error over all the points of the trajectory.

3.2.2 Machine learning for UWB localization

Robust ultra-wideband range error mitigation with deep learning at the edge (S.Angarano, V. Mazzia, F. Salvetti, G. Fantin, M. Chiaberge, 2021)[12]

This paper proposes a solution to mitigate the error given both in LOS and NLOS cases, collecting data in different situations, with different kinds of obstacles and different rooms. The channel input response (CIR) obtained from the tag is given as input to a feedforward neural network, which is used for range mitigation The ground position is collected through a total station. A focus on weight quantization,

hardware characteristic and computational lightness is then introduced. The obtained result is compared to the case without mitigation.

UWB indoor localization using deep learning LSTM networks (A. Poulose, D. S. Han, 2020)[13]

In this work, a deep learning-based system is used to calculate the position of the tag with three anchors. First, a TOA-distance model is implemented, and then the resulting ranging distances from anchors to the tag are fed as input to a two-layer LSTM neural network which estimates the position of the tag. An investigation over hyperparameters has been carried out and the best model has been evaluated in a simulation environment and compared with other techniques such as linearized least square estimation (LLSE), fingerprint estimation (FPE), maximum likelihood estimation (MLE) and weighted centroid estimation(WCE).

Feature-based deep LSTM network for indoor localization using UWB measurements (A. Poulose, D. S. Han, 2021), [14]

In this work, a method based on feature extraction on the time of arrival (TOA) data is used to feed an LSTM network which outputs the position of the receiver. The system extracts minimum, maximum and quantile at 25%, 50% and 75% from time sequences obtained from data. Then, the network is trained and the method is validated through a simulation in which anchors send Gaussian pulses. This approach is then compared with other deep learning models.

Learning-Based bias correction for time difference of arrival ultra-wideband localization of resource-constrained mobile robots (W. Zhao, J. Panerati, A. P. Schoeling, 2021) [15]

This work focuses o correcting the positive bias in the case of TDOA-based (Time Difference of Arrival) UWB localization. The proposed solution consists of a learning-based bias correction within different and unobserved anchor constellations using bias correction and M-estimation. Then, this approach is implemented on a nano-quadcopter, showing how the method can run in real-time in a closed loop on the drone board. The M-estimation technique is then used as a benchmark for the comparison of the results, in particular, the Root Mean Square Error (RMSE) is compared.

Deep gated recurrent unit-based 3D localization for UWB systems (D. Nguyen, J. Joung, X. Kang, 2021) [16]

This work proposes a 3D localization method with 8 anchors in the environment. The signals received by the receiver anchor are then downsampled and fed to a two-layer GRU neural network which outputs directly the position of the receiver. The algorithm is then simulated in different conditions of Signal to Noise Ratio (SNR) and the root mean squared error is compared with other approaches such as convolutional neural network-based (CNN) methods and trilateration.

Chapter 4

Machine learning

Since ancient times, man always dreamt about creating an autonomous thinking machine, and nowadays the technology innovation is moving in this direction. But first, to introduce a distinction between the most common terms, such as *Artificial Intelligence* (AI), *Machine Learning* (ML), *Neural networks* (NN) and *Deep Learning* (DL), in order not to confuse. According to A. Moore: "Artificial intelligence is the science and engineering of making computers behave in ways that, until recently, we thought required human intelligence". Just to mention a few, recognition, prediction, translation and making decisions are examples of machine learning. This task can be accomplished by traditional programming, written through human coding, or machine learning, which is the ability of a machine to extract its knowledge from raw data. Neural networks are a subsystem of the machine learning universe and are characterized by their structure inspired by the human brain. Deep learning is characterized by a neural network with multiple hidden layers, hence the adjective "deep". In summarising, it is possible to see this hierarchy as nested Russian dolls, each one as a subsystem of the other. [17]

The rest of the chapter contains a brief panoramic on the history of neural networks, an overview of the main elements of neural networks, including the feedforward neural network architecture, activation functions, stochastic gradient descent algorithm and ADAM optimizer, and a simple introduction to recurrent neural networks.

4.1 Brief history of neural networks

The history of deep learning is characterised by several peaks and valleys. The first attempts date back to the 1940s, from a discipline known as *cybernetics*, where a simple linear model was designed to imitate brain functions.

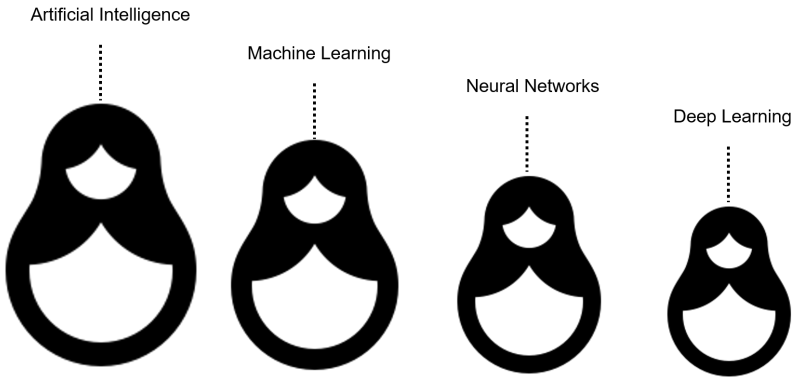


Figure 4.1: Nested russian dolls representing the AI hierarchy. [17]

The McCulloch and Pitts neuron (1943) aimed at imitating the brain cell functioning with a linear model, where n inputs $x_1, \ldots, x_n$ were associated with an output y through a linear function $f(x, w) = x_1 w_1 + \cdots + x_n w_n$, where the weights $w_1, \ldots, w_n$ were set by a human operator. Checking the sign of its output it is possible to distinguish the input into two categories. Rosenblatt's *perceptron* (1958) was the first evolution step, being able to learn the weight to divide inputs into categories, given a series of examples. Widrow's and Hoff's *adaptive linear element* (ADALINE) (1960), has a similar structure to the perceptron but is able to output a real number as $f(x)$ learning the weight from data. Its training algorithm, called *stochastic gradient descent* is still the most used nowadays. The previously listed models are called linear models and are still extensively used, with different training techniques. The first obstacle that deep learning faced was its inability to learn the XOR function, considering the simple linear models that were developed. This problem was raised in a paper by Minsky and Papert (1969) that caused a negative response against neuron-inspired learning and started the "first winter" of deep learning, a period when researchers lost interest in it.

During the 1980s a new rise in neural network research happened thanks to the *connectionism* movement, born in the field of cognitive science and psychology to explain symbolic reasoning. Its core principle is that complexity can be achieved by networking together a huge number of simple computational units. Nowadays many ideas that were developed during this "second wave" are still used. The first one is the concept of *distributed representation* by Hildon and al. (1986), which means that inputs should be represented by their shared features, inspired by symbolic reasoning. Another important achievement that is still largely in use is the *back-propagation* algorithm for neural networks with hidden layers, developed

by Rumelhart et al (1986) and LeCun (1987). The *neocognitron*, developed by Fukushima (1980) was a type of architecture used to process images and inspired the convolutional neural networks, developed by LeCun and al. (1998). Long short-term memory (LSTM) neural networks were developed by Hochreiter and Schmidhuber (1997), after the studies of Hochreiter (1991) and Bengio et al. (1994) about the mathematical difficulties of modelling long sequences; this kind of network is still used nowadays to accomplish many tasks, including the natural language process. The "second winter" of deep learning arrived in the mid-1990s when the physical hardware could not execute all the computation needed to fulfil the ambitious expectations, so investors were disappointed and removed funds from research.

The "spring" of deep learning started in 2006 when the combination of the improvement of the computing capacity and the development of a more efficient models made it possible to outperform other AI systems. Since then, the term "deep" has been emphasised to highlight the capability to train deeper networks that was not possible before. Thanks to the contribution of many researchers such as Hinton, Bengio, LeCun, Delleau, Pascanu and many others, it was possible for the deep neural network to outperform many other techniques of machine learning and hard-programmed functionalities.

Another factor that made the outbreak of deep learning possible is the availability of a large amount of data: due to the digitalization of society, it is easy to record and organize in *datasets* many human activities. A role rule of thumb suggests that, in a supervised learning algorithm, 5000 examples are enough to obtain satisfactory performance, while at least 10 million examples are needed to overcome human results. The first recorded datasets were all compiled by hand and studied by statisticians, while the first models of machine learning, around the 1950s, were trained on small and synthetic ones. Later, around the 1980s, it was possible to handle bigger datasets with thousands of samples, for example, the MNIST dataset (a collection of hand-written digits). Finally, in the 2010s, datasets consist in thousands or millions of examples, such that deep neural networks are properly trained.

The availability of faster CPUs and GPUs is one of the most important boosts in deep learning history: it makes it possible to handle deep neural networks without limitations (roughly, since the introduction of the hidden layers, they have doubled every 2.4 years).[18]

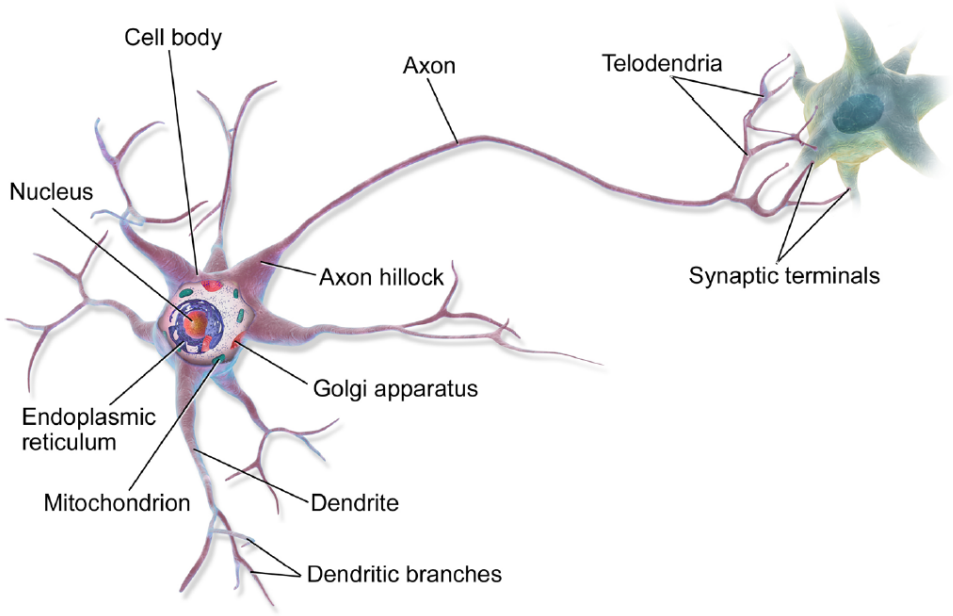


Figure 4.2: Biological neuron. [19]

4.2 Main elements of deep neural networks

The idea of the primitive computational units, such as the *perceptron* was inspired by biology, specifically from the brain cells of animals. Neurons have a rather unusual structure compared to other types of cells. They are composed of:

- the *soma*, which is the body of the cells and contains the nucleus
- the *dendrites*, extensions of the main body
- the *axon*, an extension of the body cell that can reach the length of thousands of times the width of the body cell; its task is to carry nerve signals
- the *axon terminal*, located at the end of the axon, where *synapses* are located: they are responsible for releasing neurotransmitters and are connected to the dendrites of other neurons.

Thus, roughly speaking, neurons are connected in a network-like structure and exchange signals among each other through electrochemical processes. Let's follow the path of the signal: if it arrives at the axon terminal, a certain amount of neurotransmitter is released from the synapses towards the dendrites of other neurons. Its quantity can weaken or boost the strength of the transmission, acting like a weight. This chemical is then transformed into a very small electric current

that flows from the dendrites to the cell body and, if the sum of all the currents received in a very small amount of time (in the order of milliseconds) is enough, the neuron fires its own signal and so on. This behaviour, organized in a network structure made up of billions of neurons, allows the brain to perform highly complex computations. [20]
Although this is an extremely simplified model of biological behaviour, that is still studied today, it is useful to explain the working principle of the first artificial neurons that were conceived.

4.2.1 McCulloch and Pitts neuron

The structure of McCulloch and Pitts's first artificial neuron is strongly inspired by the biological model. The n inputs $x_1, \ldots, x_n$ are constituted by signals that can assume only binary values. Inputs can either be *inhibitory* or *excitatory*. Inhibitory inputs have the strongest effect on the output: if one of them is equal to zero, the output will be equal to zero, while excitatory inputs are multiplied by a weight w_i and summed; if the sum is greater than a threshold θ, the output will be 1, otherwise it will be equal to 0. [21]
The behaviour of this neuron can be described by the following formula.

$$y = \begin{cases} 1 & if \quad \sum_{i=1}^{n} w_i x_i > \theta \quad \wedge \quad !inhibiton \\ 0 & if \quad inhibition \end{cases} \tag{4.1}$$

Complex logical expressions can be computed if many artificial neurons are connected to form a network. [20]

4.2.2 The perceptron

The perceptron, also known as *threshold logic unit* (TLU), was invented in 1957 by F. Rosenblatt and it is one of the simplest ANN architectures, it represents an evolution of McCulloch's and Pitts's artificial neuron. The main innovation was an algorithm able to train iteratively the parameters of the neurons from data. The main differences are that there are no inhibitory inputs and every neuron has different positive and negative weights and biases.
Given a set of N examples, let's suppose it is possible to distinguish them into two classes, by their n input features. So, it is possible to identify a hyperplane $\mathcal{H}$ that can divide the examples into two classes.

$$\mathcal{H} = \{x \in \mathbb{R}^n : w^T x + b = 0\} \tag{4.2}$$

Therefore, $\mathcal{H}$ separates the space $\mathbb{R}^n$ in two subspaces $\mathcal{H}^+$ and $\mathcal{H}^-$.

$$\mathcal{H}^+ = \{x \in \mathbb{R}^n : w^T x + b \geq 0\} \qquad \mathcal{H}^- = \{x \in \mathbb{R}^n : w^T x + b < 0\} \tag{4.3}$$

So, it is possible to discriminate the features based on $\mathcal{H}$ as a discrimination surface: the hyperplane $\mathcal{H}$ is therefore a *decision boundary* and the output of the perceptron can be a binary value, depending on the class the input is associated to. The thresholding function can output values in $[0, 1]$ or $[-1, 1]$, depending if a Heaviside step function (4.4) or a sign function (4.5) is chosen.

$$heaviside(z) = \begin{cases} 1 & if \quad z \geq 0 \\ 0 & if \quad z < 0 \end{cases} \tag{4.4}$$

$$sign(z) = \begin{cases} +1 & if \quad z > 0 \\ 0 & if \quad z = 0 \\ -1 & if \quad z < 0 \end{cases} \tag{4.5}$$

Thus, the final formula of the single neuron perceptron with a bias term b, using the sign function can be written as in (4.6); in case the Heaviside function is used, it can be derived similarly.

$$y = \begin{cases} 1 & if \quad \sum_{i=1}^{n} w_i x_i + b \geq 0 \\ -1 & otherwise \end{cases} \tag{4.6}$$

Thus, the objective of the learning algorithm of the perceptron is to find a hyperplane $\mathcal{H}$ that minimizes the distance of misclassified points to the decision boundary. If a sample that should be classified with $y^{(i)} = 1$ is misclassified, then $x^{(i)}w + b < 0$, and viceversa for misclassified $y^{(i)} = -1$.
Thus, it is possible to define a cost function $L(w, b)$ that needs to be minimized.

$$L(w, b) = \sum_{i=1}^{N} [-y_i(x_i w + b)]_+ \tag{4.7}$$

In formula this formula the $[\cdot]_+$ operator is defined as:

$$[x]_+ = \begin{cases} x & if \quad x > 0 \\ 0 & otherwise \end{cases} \tag{4.8}$$

Thus, the contribution in the cost function for the correctly classified points is null, while it is proportional to their distance from the decision boundary otherwise. The cost function $L(w, b)$ is *piece-wise linear* and can be minimized iteratively. [22]

Algorithm 1 Perceptron's algorithm

$w = 0, b = 0$
while not all classified well **do**
  if $-y_i[w_i x_i + b] < 0$ **then**
    $w \leftarrow w + y_i x_i$
    $b \leftarrow b + y_i$
  end if
end while

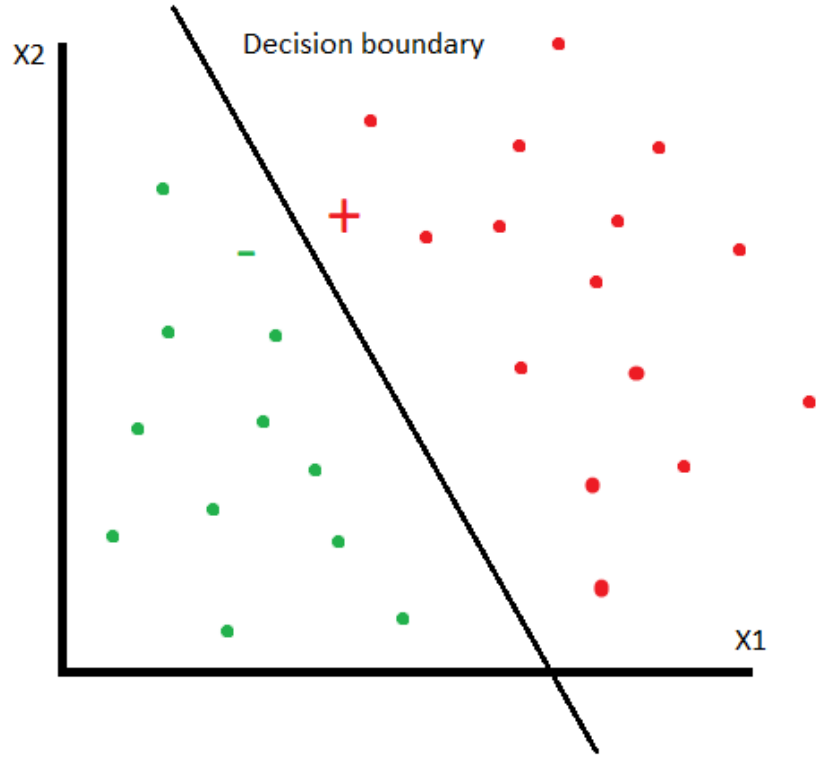


Figure 4.3: Example of decision boundary in binary classification problem.

4.2.3 Feedforward neural networks architecture

Feedforward neural networks, also called *multilayer perceptrons*, are composed of multiple layers of neurons. The first layer is called the *input layer*, the last one *output layer* and the ones in the middle are *hidden layers*. The layers near the input are called *lower layers*, while the ones near the output, *upper layers*. Each layer is fully connected with the next one, meaning that the outputs of all the neurons of a layers are the inputs of all the neurons in the next one. This kind of network is called feedforward because the signals only flow in one direction and no loops are present.

The following notation will be used

- n_ℓ: number of layers
- L_ℓ: ℓ-th layer, with $\ell = 1, \dots, n_\ell$
- $(W, b) = (W^{(1)}, b^{(1)}, \dots, W^{(n_\ell - 1)}, b^{(n_\ell - 1)})$: parameters of the network

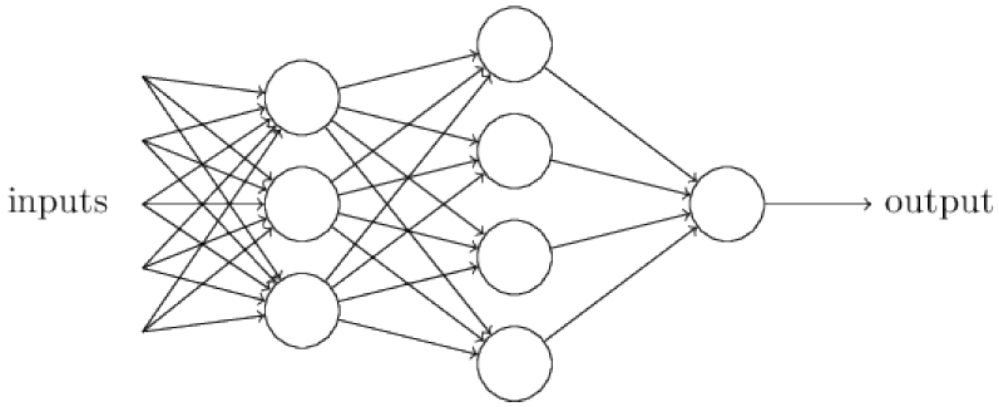


Figure 4.4: Example of a structure of a feedforward NN. [23]

- $W_{ij}^{(ell)}$: weight associated with the link between unit j in layer ℓ and unit i in layer $\ell + 1$
- $b_i^{(\ell)}$: bias associated with unit i in layer $\ell + 1$
- s_ℓ: number of nodes in layer ℓ (not counting the bias)
- $a_i^{(\ell)}$: the activation, or output value, of unit i in layer ℓ.
- $h_{W,b}(x)$: hypothesis, output of the neural network

For all the layers $\ell = 1, \ldots, n_\ell - 1$ it is possible to write the shapes of the matrices.

$$W^\ell \in \mathbb{R}^{s_{\ell+1}, s_\ell} \tag{4.9}$$

$$b^\ell \in \mathbb{R}^{s_{\ell+1}} \tag{4.10}$$

If $\ell = 1$, it is possible to write $a_i^{(1)} = x_i$ to indicate the i-th input, so it is possible to derive a general formula to calculate activation at any layer, using a *non-linear recursion.* [24]

$$z^{(\ell+1)} = W^{(\ell)} a^{(\ell)} + b^{(\ell)} \tag{4.11}$$

$$a^{(\ell+1)} = f(z^{(\ell)}) \tag{4.12}$$

4.2.4 Activation functions

Considering the learning principle, which consists in making small adjustments in the value of the weights depending on the discrepancy between the network's output and the desired output, in order to finely tune the parameters of the network to obtain greater accuracy, it would be great that a small change in the weights and biases causes only a small change in the output. In the case of perceptron, where the activation function is a Heaviside step or a sign function, a very little alteration of the parameters can lead the output to change switch, for example from 1 to -1. This phenomenon can lead the network to behave in a complicated way and to unexpected consequences. Later on, the most important activation functions will be briefly introduced.[23]

Sigmoid function

In order to overcome this problem, the *sigmoid* (or *logistic*) function can be used as an activation function. It is smooth, differentiable and saturates at 0 for $z \to -\infty$ and at 1 for $x \to +\infty$. The only problem is that it is hard to compute.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{4.13}$$

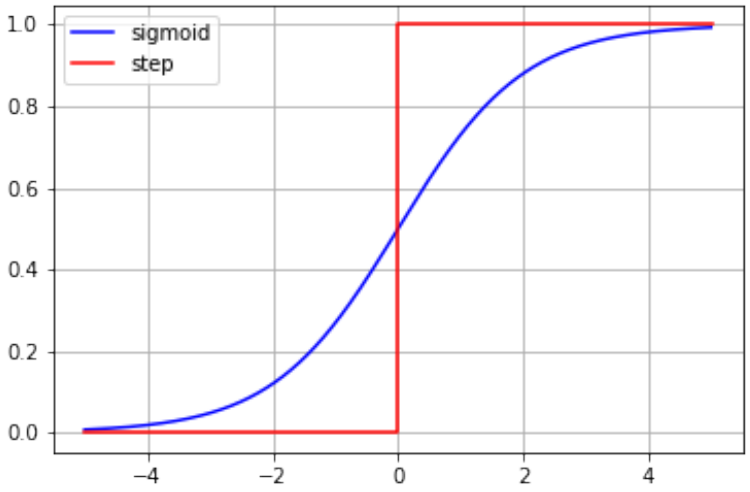


Figure 4.5: Sigmoid and Heaviside step function compared.

Hyperbolic tangent

The *hyperbolic tangent* has the same features as the logistic function, except that it saturates at -1 for $z \to -\infty$. They can be used equivalently, depending on the application.

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{4.14}$$

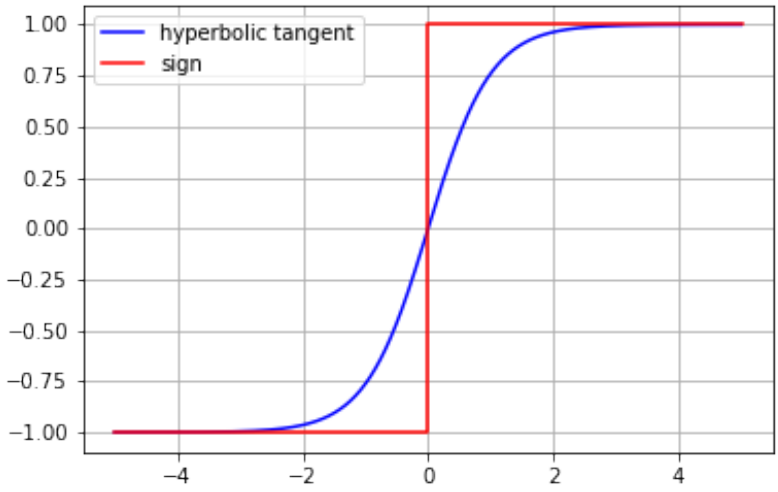


Figure 4.6: Hyperbolic tangent and sign function compared.

Rectified linear unit

The *rectified linear unit function* (ReLU), defined basically as a linear ramp, is one of the most popular activation functions in feedforward neural networks. A linear function would make the network lose all the non-linear properties transforming it into a linear function itself, whereas the ReLU's output yields a nonlinear transformation. Being, in reality, made of two linear pieces, it preserves many properties of linear units such as the easy optimization with gradient descent based methods, the capacity of generalization and the computation lightness. [18]

$$ReLU(z) = max(0, z) \tag{4.15}$$

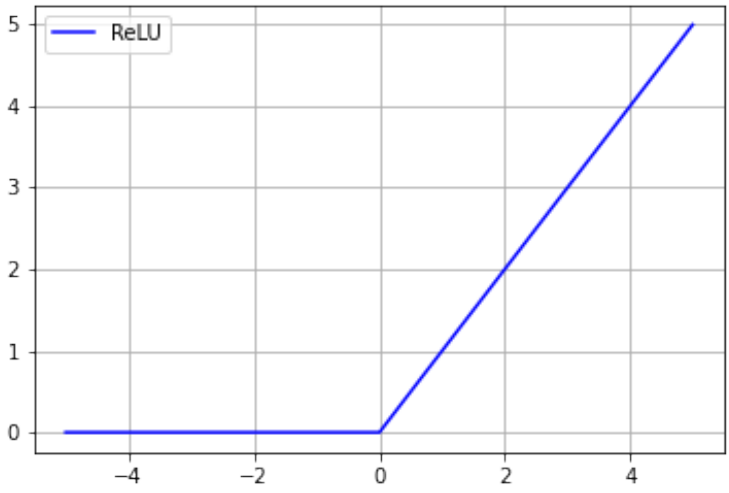


Figure 4.7: ReLU function.

Softmax

The *softmax* activation function is generally used as an output layer in case of classification problems. Summing all the activation of this layer they will be equal to one: it can be seen as a probability function for each class represented by each output neuron j.

$$softmax(z_j) = \frac{e^{z_j}}{\sum_j e^{z_j}} \tag{4.16}$$

4.2.5 Gradient descent algorithm

In order to properly train the network, we need to define a function that quantifies the discrepancy e between the output value $h_{W,b}(x)$, given an input x, and the expected value y.

$$e_{W,b}(x) = h_{W,b}(x) - y \tag{4.17}$$

We can define the error over a set of m training examples $(x_1, y_1), \ldots, (x_m, y_m)$ and define an overall measure of mismatch. A common choice is the *Mean Squared Error* (MSE), which in the case of scalar output is in the form:

$$MSE(W,b) = \frac{1}{m}\sum_{t=1}^{m}\frac{1}{2}\left|e_{W,b}(x^{(t)}, y^{(t)})\right|^2 = \frac{1}{m}\sum_{t=1}^{m}\frac{1}{2}(h_{W,b}(x^{(t)}) - y^{(t)})^2 \tag{4.18}$$

Another option is the *Mean Absolute Error* (MAE).

$$MAE(W,b) = \frac{1}{m}\sum_{t=1}^{m}\left|e_{W,b}(x^{(t)},y^{(t)})\right| = \frac{1}{m}\sum_{t=1}^{m}\left|h_{W,b}(x^{(t)}) - y^{(t)}\right| \tag{4.19}$$

In the case of multiple outputs, for the MSE the formulation becomes

$$J(W,b) = \frac{1}{m}\sum_{t=1}^{m}\frac{1}{2}||_{W,b}(x^{(t)}) - y^{(t)}||_2^2 \tag{4.20}$$

Since the objective is to predict the values on unseen data, in order to prevent *overfitting* it is common to add a regularization term on the magnitude of the weights that reduces their magnitude; λ controls the relative importance of the two terms. [24]

$$J(W,b) = \frac{1}{m}\sum_{t=1}^{m}\frac{1}{2}||_{W,b}(x^{(t)}) - y^{(t)}||_2^2 + \frac{\lambda}{2}\sum_{\ell}^{n_\ell - 1}\sum_{i=1}\sum_{j=1}|W_{ij}^{(\ell)}|^2 \tag{4.21}$$

Considering the form of the quadratic cost function (4.20) without the weights regularization, we see that $J(W,b)$ is non-negative since all the terms are non-negative. Moreover, its value gets smaller as $h_{W,b}$ approximates the expected value for all the training examples. Therefore, the objective of the training is to *minimize* the value of the cost function, which means in other words to find a set of parameters which makes its value as small as possible for obtaining the best predictions. It can be done through the *gradient descent algorithm* and this cost function suits perfectly for this task: a function of the parameters is chosen to understand how to change them to obtain better results.

Let's step out from the specific problem of neural networks for a while and suppose we want to minimize a real-valued function of two variables $f(z_1, z_2)$, where z_1, z_2 are the variables. The first option is to use calculus and find the minimum analytically, which works great in cases like this, where there are only a few variables, but it becomes computationally impossible when there are millions or billions, like in the case of neural networks. Another option is the gradient descent algorithm. Since it is still possible to visualize a two variables function, it can be imagined as a valley-shaped surface with an *absolute minimum*. Let's imagine putting a ball at a random point and simulating its descent to the bottom due to gravity: it can be done just by computing derivatives to know the local shape of the surface. The objective, in this case, is to choose a law to make the ball always roll to the bottom of the valley. We can start by writing that, if we move the ball of Δz_1 along z_1 direction and Δz_2 along z_2 direction, the function changes by Δf.

$$\Delta J \approx \frac{\partial J}{\partial z_1}\Delta z_1 + \frac{\partial J}{\partial z_2}\Delta z_2 \tag{4.22}$$

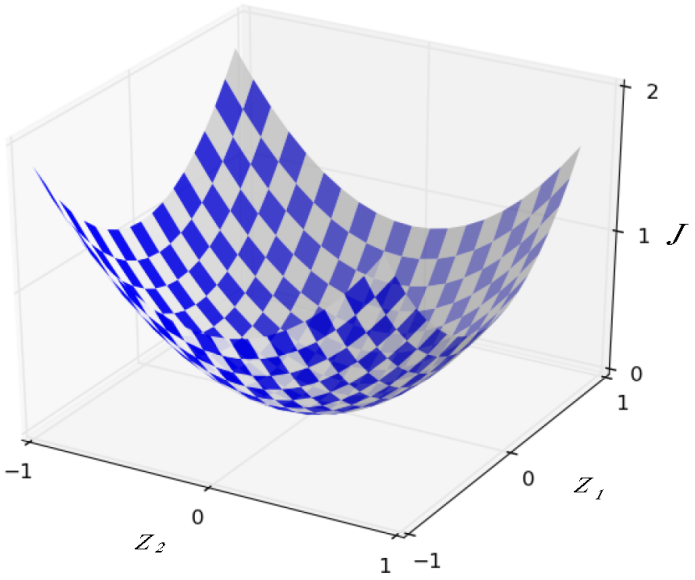


Figure 4.8: $f(z_1, z_2)$ [23]

So we need to find the value of $\Delta z \equiv (\Delta z_1, \Delta z_2)$ in order to make Δf negative. We also define the ∇ as the gradient of the function.

$$\nabla J \equiv \left(\frac{\partial J}{\partial z_1}, \frac{\partial J}{\partial z_2} \right)^T \tag{4.23}$$

So that we can use the compact notation and rewrite the formula (4.22).

$$\Delta J \approx \nabla J \cdot \Delta z \tag{4.24}$$

Given this equation, it is possible to choose $\Delta z = \eta \nabla J$ where η is called the *learning rate*. Replacing, we obtain $\Delta J \approx -\eta \nabla J \cdot \nabla J = -\eta ||\nabla J||^2$ Being the square of the gradient $||\nabla J||^2$ equal to or greater than zero, it is guaranteed that $\Delta J \leq 0$, which means that J will never increase if we change z according to this rule. So the 'law of motion' of our ball will be:

$$z \to z' = z - \eta \nabla J \tag{4.25}$$

Applying it iteratively will bring the ball to the global minimum. Obviously, this model does not take into account the friction and the momentum of the ball, but only the necessary elements for the sake of the explanation of the gradient descent. In order to make this algorithm work properly, a suitable value for the learning rate η should be chosen. Its value has to be small enough in order not to

make $\Delta J > 0$ according to (4.24), which would cause the algorithm's divergence. Concurrently, it should not be too small, which would lead to very slow convergence.

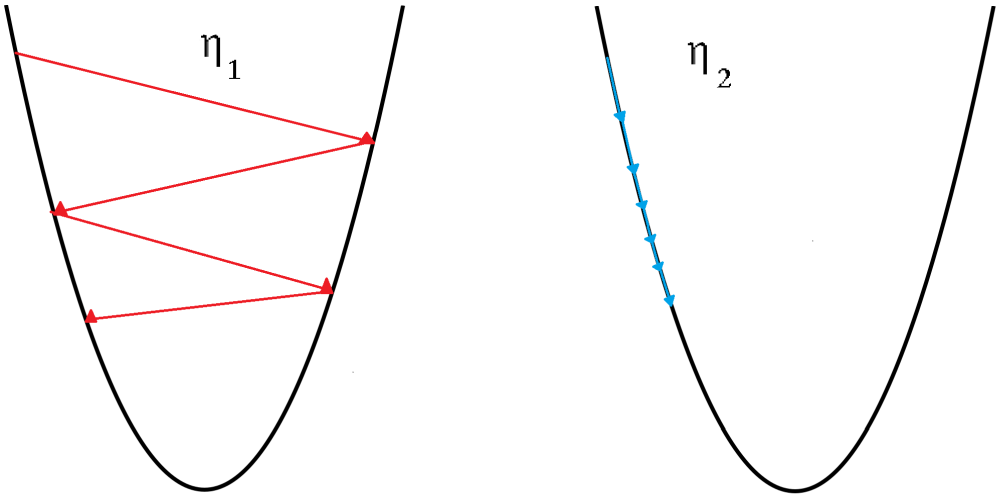


Figure 4.9: On the left the learning rate η_1 is too big and does not lead to convergence. On the right η_2, slowly leads to convergence.

We can extend this working principle to m variables $z_1, \ldots, z_m$ using the same formulation. Therefore, it is possible to apply the same update rule with great results: it turns out to be a powerful method to minimize a convex cost function J .

Therefore, it is possible to apply the same gradient descent algorithm to the neural network's cost function $J(W, b)$ as a way to find the best weights w and biases b that minimize it. So now the gradient vector has components $\frac{\partial J}{\partial W_{ij}^{\ell}}$ and $\frac{\partial J}{\partial b_i^{\ell}}$ and it is possible to apply the same update rule.

$$W_{ij}^{\ell} \leftarrow W_{ij}^{\ell} - \eta \frac{\partial J(W, b)}{\partial W_{ij}^{\ell}} \tag{4.26}$$

$$b_i^{\ell} \leftarrow b_i^{\ell} - \eta \frac{\partial J(W, b)}{\partial b_i^{\ell}} \tag{4.27}$$

4.2.6 Stochastic gradient descent

Looking at the structure of the input, it has the form $J(W, b) = \frac{1}{m} \sum_{x_j} J_{x_j}$, that is essentially an average over the cost function over the m input examples. This requires a huge number of computation, which can make the training very slow. The

stochastic gradient descent algorithm was devised in order to reduce the training time, requiring less calculation. It consists in choosing at each training step a random subset, or *mini-batch* of n elements, of training inputs, $X_1, \dots X_n$ large enough to be representative for all the input examples. It follows that the average value of ∇J_{X_j} will approximate the average value of ∇J_x over all the input samples.

$$\frac{1}{n}\sum_{j=1}^{n}\nabla J_{X_j} \approx \frac{1}{m}\sum_{i=1}^{m}\nabla J_{x_i} = \nabla J \tag{4.28}$$

Thus, it is possible to apply the same rule directly to the update rule of neural network's weights and biases.

$$W_{ij}^{\ell} \leftarrow W_{ij}^{\ell} - \frac{\eta}{m}\sum_{X}\frac{\partial J_X(W,b)}{\partial W_{ij}^{\ell}} \tag{4.29}$$

$$b_{i}^{\ell} \leftarrow b_{i}^{\ell} - \frac{\eta}{m}\sum_{X}\frac{\partial J_X(W,b)}{\partial b_{i}^{\ell}} \tag{4.30}$$

In summary, in order to train a neural network:

- a minibatch is chosen among the training inputs
- the weights are updated with the update rule

until all the training samples are used. this is called an *epoch* of training. Then, another training epoch starts again. [23]
The number of epochs required to train a neural network may change according to the network size, mini-batch dimension, learning rate and many other parameters. If a mini-batch with a size equal to one is chosen, the learning process is called *online* or *incremental* and can be done directly while data are generated, similarly to human learning.

4.2.7 Backpropagation

Once we have defined a cost function and the gradient descent algorithm, needed for training, it is necessary to find a way to compute the gradient of the cost function and *backpropagation* is a solution for this problem. In a few words, it consists in evaluating how the modification of the value of weights and biases can change the cost function. We consider the quadratic cost function, also called mean square error, in the form (4.20), such that it was written as an average $J = \frac{1}{m}\sum_x J_x$ over the cost function of the individual training samples and it is written as a function of the outputs of the network. This is necessary because backpropagation evaluates the partial derivatives for the single examples and then averages them.

We define δ_j^ℓ as the error of the j-th neuron in ℓ-th layer: it is possible to compute it through backpropagation and relate it to the partial derivatives $\frac{\partial J}{\partial W_{ij}^\ell}$ and $\frac{\partial J}{\partial b_i^\ell}$. To better understand how backpropagation works, let's imagine to apply a little change Δz_j^ℓ to the weighted input of a neuron, such that its output is equal to $\sigma(z_j^\ell + \Delta z_j^\ell)$ instead of $\sigma(z_j^\ell)$. Evaluating the output, given the small change Δz_j^ℓ, the overall cost function will be altered by a value equal to $\dfrac{\partial J}{\partial z_j^\ell}\Delta z_j^\ell$. The objective is to find a Δz_j^ℓ that reduces the value of the cost function: if $\dfrac{\partial J}{\partial z_j^\ell}$ has a large value, it can be decreased by choosing Δz_j^ℓ with the opposite sign. If, on the contrary, the value of $\dfrac{\partial J}{\partial z_j^\ell}$ is near zero, it can not be improved a lot. In conclusion, the error δ_j^ℓ can be defined as:

$$\delta_j^\ell \equiv \frac{\partial J}{\partial z_j^\ell} \tag{4.31}$$

Backpropagation offers a way to compute δ^ℓ for every layer and relate it to $\frac{\partial J}{\partial W_{ij}^\ell}$ and $\frac{\partial J}{\partial b_i^\ell}$: the algorithm consists in four main equations that are required to compute both the error and the gradient of the cost function.

Equation 1.

$$\delta_j^L = \frac{\partial J}{\partial a_j^L}\sigma'(z_j^L) \tag{BP1}$$

The equation allows evaluating the error for the output layer δ^L. The $\dfrac{\partial J}{\partial a_j^L}$ part calculates how the cost function changes as a function of the activation of the j-th output, the $\sigma'(z_j^L)$ part measures how the activation function $\sigma()$ changes at a variation of z_j^L. It is possible to write the matrix form of (BP1) equation as in the following formula, where $\odot$ indicates the element-wise operation.

$$\delta^L = \nabla_a J \odot \sigma'(z^L) \tag{BP1a}$$

Where ∇_a is the vector of partial derivatives of $\dfrac{\partial J}{\partial a_j^L}$.

Equation 2.

$$\delta^\ell = ((W^{\ell+1})^T\delta^{\ell+1}) \odot \sigma'(z^\ell) \tag{BP2}$$

This equation evaluates the error δ^ℓ as a function of the error of the next layer: this allows moving the error 'backwards' through the network. Combining this equation and (BP1) it is possible to measure the error δ^ℓ for every layer.

Equation 3.

$$\frac{\partial J}{\partial b_i^\ell} = \delta_i^\ell \tag{BP3}$$

This equation measures the rate with which the cost function varies with respect to any bias.

Equation 4.

$$\frac{\partial J}{\partial W_{ij}^\ell} = a_j^{\ell-1}\delta_i^\ell \tag{BP4}$$

This equation measures the rate with which the cost function varies with respect to any weight.

Given all the four equations, it is possible to explicitly write the backpropagation algorithm.

1. **Input:** set the input x as activation a^1 of the input layer.
2. **Feedforward:** for each layer $\ell = 2, \ldots, n_\ell$ compute $z_\ell = W^\ell a^{\ell-1} + b^\ell$ and $a^\ell = \sigma(z_\ell$.
3. **Output error:** compute δ^L using equation (BP1a).
4. **Backpropagate the error:** for each layer δ^ℓ is computed using (BP2).
5. **Output:** the gradient of the cost function with components $\dfrac{\partial J}{\partial W_{ij}^\ell}$ and $\dfrac{\partial J}{\partial b_i^\ell}$ is computed using (BP3) and (BP4).

From the formulation of the algorithm, the origin of the name backpropagation is clear: the error, as a function of the outputs, is propagated until the lower layers after the output is evaluated from the input, therefore the training process consists in a back-and-forth movement through its structure. [23]

4.2.8 ADAM optimizer

The *ADAM* optimization algorithm, which stands for *adaptive moment estimation*, is a first-order gradient-based optimization algorithm for stochastic objective functions; it is based on estimating *low-order moments*. Even though stochastic gradient descent (SGD) is still the most popular algorithm for deep learning training and optimization, ADAM is gaining popularity due to its straightforward implementation, computational efficiency and memory requirements. Moreover, this algorithm will be used in this work.

The ADAM algorithm combines the advantages of the *AdaGrad* method (Duchi et al., 2011), which is suitable for sparse gradients, and *RMSProp* (Tieleman and Hildon, 2012), which is suitable for online and non-stationary settings. ADAM needs only first-order gradients, so only a few memory is needed; moreover, the magnitude of parameter update is invariant to gradient rescaling and stepsize is bounded by a hyperparameter.

Before explaining the algorithm working principle, let's introduce the concept of the moment of order n of a random variable X: it is defined as the expected value of the variable at the power of n, namely $\mathcal{E}[X^n]$. Then, let $f(\theta)$ be the noisy objective function to minimize, differentiable with respect to the parameters θ: the first moment of this function can be defined as the mean of its realizations, while the second moment as its variance. Defining the gradient of the current mini-batch as $g_t = \nabla_\theta f_t(\theta)$, the algorithm consists in updating the moving average m_t and variance v_t.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{4.32}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{4.33}$$

Where β_1 and β_2 are constant hyperparameters in the interval $[0, 1)$, which control the exponential decay rates of the moving averages. Their usual value is typically $beta_1 = 0.9$ and $\beta_2 = 0.999$. Since the moving averages are initialized equal to zero, this leads the moments to be biased towards zero, but this can be easily corrected by obtaining the values $\hat{m}_t$ and $\hat{v}_t$.

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{4.34}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{4.35}$$

Therefore, defining the learning rate as $\eta_t = \eta \cdot \frac{1-\beta_2^t}{1-\beta_1^t}$, it is possible to define the update rule.

$$\theta_t \leftarrow \theta_{t-1} - \eta_t \cdot \frac{m_t}{\sqrt{v_t} + \epsilon} \tag{4.36}$$

The complete ADAM algorithm from the original paper from D. P. Kingma and J. L. Ba [25] is reported below.

Algorithm 2 ADAM optimization algorithm

Require: α: stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: decay rates for moment estimation
Require: $f(\theta)$: stochastic objective function with θ parameters
Require: θ_0: initial parameter vector
  $m_0 \leftarrow 0$ ▷ (1^{st} moment estimate set to 0)
  $v_0 \leftarrow 0$ ▷ (2^{nd} moment estimate set to 0)
  $t \leftarrow 0$ ▷ (timestep set to 0)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ ▷ compute gradients with respect to parameters θ at t
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ ▷ (update of biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ ▷ (update of biased second raw moment estimate)
    $\hat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t}$ ▷ bias-correction of first moment estimate
    $\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t}$ ▷ bias-correction of raw second moment estimate
    $\theta_t \leftarrow \theta_{t-1} - \frac{\eta \cdot \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$ ▷ (update of parameters)
  end while
  return θ_t ▷ (resulting parameters)

4.3 Recurrent neural networks

Until now in the dissertation on the neural network fundamentals only feedforward neural networks have been examinated: for the sake of the objective of this thesis, since temporal sequences are considered, the fundamentals of the *recurrent neural networks* (RNN) will be studied. Thus, in contrast to multilayer perceptrons where only connections towards a higher layer were allowed, in recurrent neural networks, cyclical feedback connections are present. This leads to the capacity of RNN to

process temporal sequences with a memory from the previous time steps that persists in the internal network's state. [26]

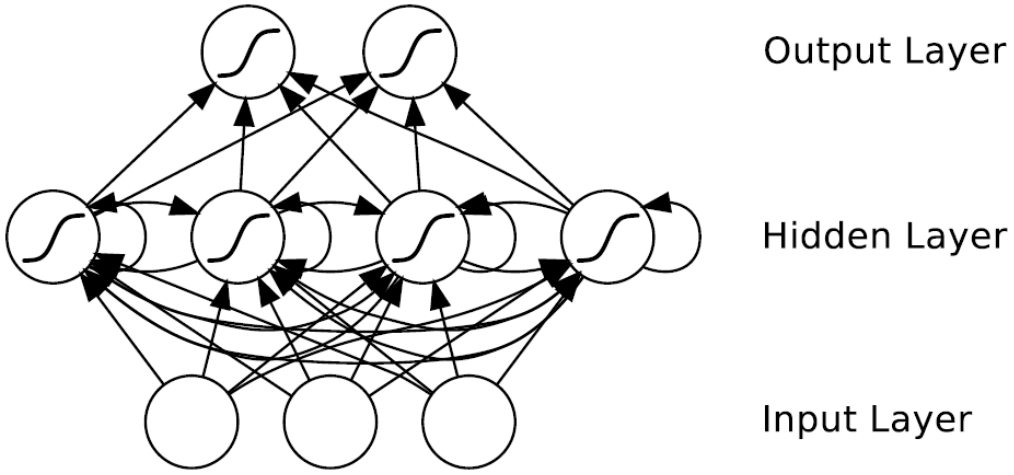


Figure 4.10: Structure of a recurrent neural network. [26]

A recurrent neural network can take a sequence as input and produce another one as output: in this case, the network is denominated *sequence-to-sequence* network; it can be used to predict series such as stock prices or trajectories. In other cases, even if the network is fed with a sequence of inputs, only the last output is taken into consideration: this model is called a *sequence-to-vector* network. Moreover, the same input can be fed to the network over and over again at each time step and the network will predict a sequence, this kind of network is referred to as *vector-to-sequence*. Finally, it is possible to connect a vector-to-sequence network right after a sequence-to-vector, called respectively *decoder* and *encoder*: this structure allows to do translation from one language to another.[20]

4.3.1 Simple RNN working principle

It is possible to create a layer of recurrent neurons that receives, at each time step t both the input x_t and its outputs from the previous instant a_{t-1}; this requires that each neuron has two weights w_x and w_y. We can write in matrix form the two sets of weights of all the layers as W_x and W_y. Therefore it is possible to compute the output of the neuron similarly to the multilayer perceptron, just adding the *feedback* contribution.

$$a_t = f(W_x^T x_t + W_y^T a_{t-1} + b) \tag{4.37}$$

It can be said that the output of the single neuron is a function of all the inputs in the previous time steps: it constitutes a sort of internal memory of the cell and is stored in a state variable, denoted as h_t that is a function of the input for the present and previous time steps. The output of a memory cell is, then, a function of the current input and previous output. It is possible to evaluate recursively the value of the hidden state value starting from $t = 1$. At $t = 0$ generally the value is set to zero.

To train an RNN the backpropagation algorithm can be used, in the same way as the multilayer perceptron; the only difference is that the network has to be unfolded in time. This variation of the original algorithm is called *backpropagation through time* (BPTT).

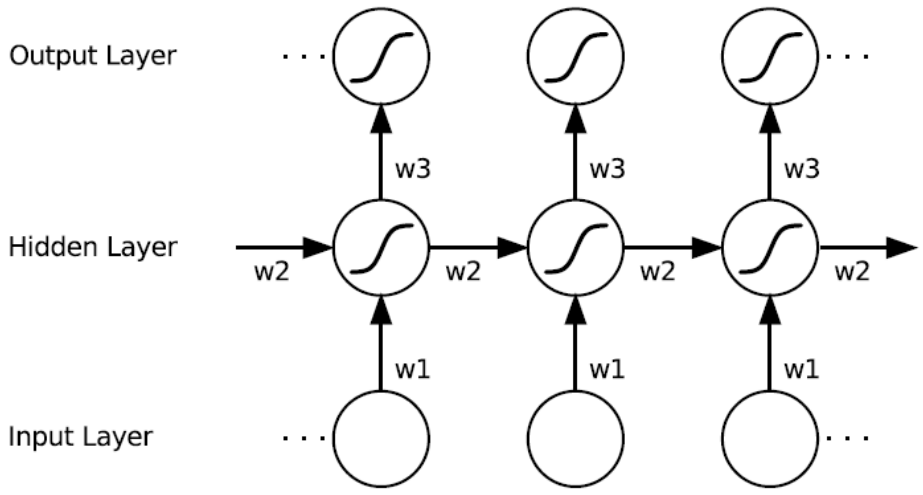


Figure 4.11: An unfolded neural network: each node represents the same layer in different time steps. It can be noticed that the weights are the same. [26]

As in the common backpropagation, as a first thing, the output of the network is evaluated given an input sequence with a length of T time instants. The cost function $J(a_1, a_2, \ldots, a_T)$ is then evaluated, considering all the output sequences, if the sequence-to-sequence case is considered, otherwise in the case of sequence-to-vector only the last one is considered. Since the network is unrolled for along time, always the same weight is considered for every time instant. [26]

4.3.2 The vanishing/exploding gradients problem

If long sequences are used to train a recurrent neural network, it is necessary to unroll it many times making it a very deep network subjected to unstable gradient problems and may take a lot of time to train. It is shown that in deep neural networks, the gradient gets smaller as the backpropagation algorithm moves forward to the lower layers, not allowing their weight to be trained properly. This is called the vanishing gradient problem. In some cases, the opposite situation may happen: the gradients' values tend to be infinite causing the exploding gradients problem. It was later found, in a paper by X. Glorot and Y. Bengio [27] that using the sigmoid activation function combined with a random weight distribution with zero mean and standard deviation equal to one may cause this problem. They have shown how the variance of the outputs of a layer with those characteristics is greater than the one of its inputs. So, when a signal goes forward in the network, its variance tends to grow until it causes the saturation of the activations of the last layers. The logistic function worsens this issue because its mean value is equal to 0.5 and not 0 as the hyperbolic tangent function. [20]

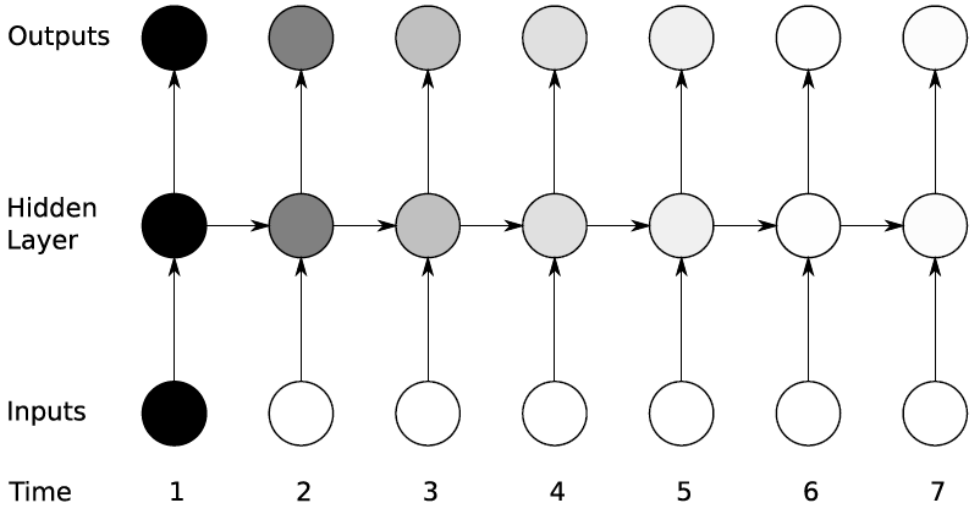


Figure 4.12: The vanishing gradient problem in RNN. [26]

4.3.3 Long short-term memory neural networks

A solution to the vanishing gradients problem and the loss of information due to the length of the sequence is represented by *Long Short-Term Memory* (LSTM) cells. From its name, it can be deduced that it tackles both the *long-term memory*

problem and the *short-term* one. If considered as a black box, an LSTM cell is similar to a classic RNN one, it has one input gate x_t, and an output gate y_t and the only difference is that the state is split into two vectors, h_t and c_t. h_t can be considered the *short-term state* and c_t as the *long-term* one.

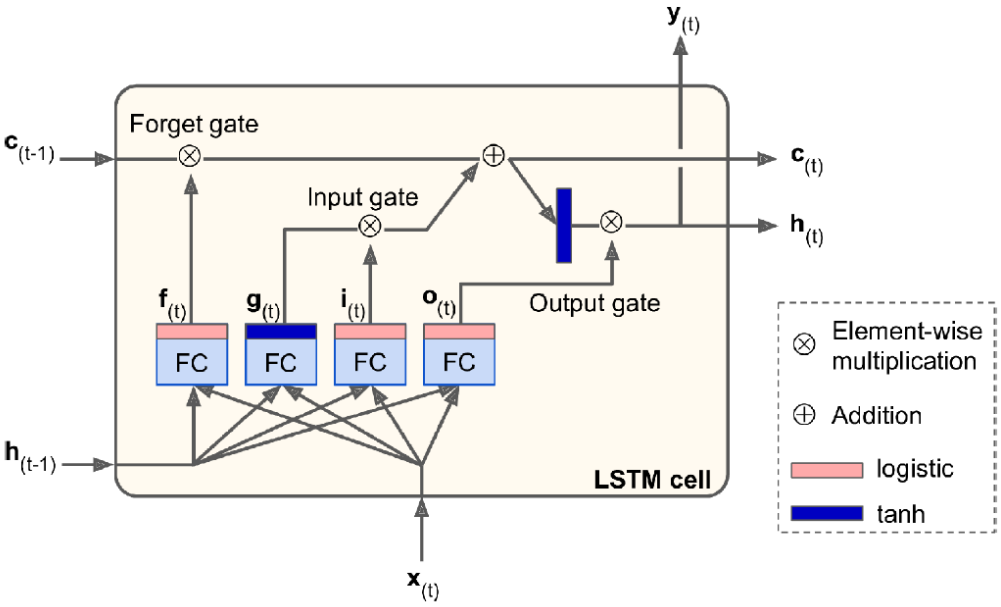


Figure 4.13: The internal structure of a Long Short-Term Memory neuron. [20]

As it can be seen in figure 4.13, the input vector $x_{(t)}$ and the short-term memory state $h_{(t-1)}$ from the previous step are the input of the cell and are fed to four different fully connected layers, with four different goals: the first one goes directly in the outputs $y_{(t)}$ and $h_{(t)}$, the other three are gate controllers.

- $g_{(t)}$ is the main one and analyzes the current input $x_{(t)}$ and the previous short-term state $h_{(t-1)}$.
- $f_{(t)}$ controls the *forget gate*, which establishes which part of the long-term state $c_{(t-1)}$ should be removed.
- $i_{(t)}$ controls the *input gate*, which establishes which part of $g_{(t)}$ should be added to the long-term state $c_{(t)}$
- $o_{(t)}$ controls the *output gate*, which establishes which part of the long-term state $c_{(t)}$ should be output to $y_{(t)}$ and $h_{(t)}$.

The long-term state $c_{(t-1)}$ goes through a forget gate, where it loses some memory and is updated with information selected by an input gate. The resulting $c_{(t)}$ is used as an output of the cell. The short-term state $h_{(t)}$ output is generated by the filtering of the hyperbolic tangent of the long-term state through an output gate. The output $y_{(t)}$ of the cell is equal to $h_{(t)}$. [20]

The main feature of an LSTM cell is the ability to recognize the significant inputs and store them in the long-term state.

The following equations summarise the behaviour of the LSTM cell.

$$i_{(t)} = \sigma(W_{xi}^T x_{(t)} + W_{hi}^T h_{(t-1)} + b_i) \tag{4.38}$$

$$f_{(t)} = \sigma(W_{xf}^T x_{(t)} + W_{hf}^T h_{(t-1)} + b_f) \tag{4.39}$$

$$o_{(t)} = \sigma(W_{xo}^T x_{(t)} + W_{ho}^T h_{(t-1)} + b_0) \tag{4.40}$$

$$g_{(t)} = tanh(W_{xg}^T x_{(t)} + W_{hg}^T h_{(t-1)} + b_g) \tag{4.41}$$

$$c_{(t)} = f_{(t)} \otimes c_{(t-1)} + i_{(t)} \otimes g_{(t)} \tag{4.42}$$

$$y_{(t)} = y_{(t)} = o_{(t)} \otimes tanh(c_{(t)}) \tag{4.43}$$

Where:

- W_{xi}, W_{xf}, W_{xo} and W_{xg} are the weights matrices of the fully connected layers with the $x_{(t)}$ input.
- W_{hi}, W_{hf}, W_{ho} and W_{hg} are the weights matrices of the fully connected layers with the $h_{(t-1)}$ input.
- b_i, b_f, b_o and b_g are the biases vectors.

Sometimes, it is better to use as input to the connected layers also the long-term state $c_{(t-1)}$ in order to give them a 'bit more context' about the past of the sequence. This extra connections are called *peephole connections*. [20]

Chapter 5

Robot platform

This chapter is devoted to illustrating to the reader the experimental instruments used during the development of this work. Since wheeled robots are just a small part of the robotic universe, the first section offers a simple panoramic classification of the main categories of mobile platforms and a simple kinematic model for differential drive robots. The rest of the chapter describes in detail the sensor used for the data collection, the used mobile robot and the software employed in the development of the work.

5.1 Introduction to mobile robots

The design of a mobile robot involves a consistent variety of knowledge fields, thus every aspect of this process can be faced by different disciplines with the specific aim of accomplishing a specific task: it can embrace mechanics, automatics, computer science, bio-engineering and electronics. [1]
Two main components can be identified in a robot: the *hardware*, that consists in all the mechanical joints, links, actuators, sensors and the electronics, and the *software*, that controls the robot. Since robots are used for many different applications, they can be classified in many categories that can partially overlap. [28]

- *Industrial robots*, which mimic the structure of a human arm and wrist, made by a rigid mechanical structure made by a chain of *joints* and *links*. They can handle high payloads and precision tasks.
- *Service and exploration robots*, which are made to work in different environments: *indoor* (flat surfaces) and *outdoor* (land, air, water) and can be divided in to three main categories based on their locomotion system:
    1. *Wheeled robots* or *UGV (Unmanned Ground Vehicles*, which are characterized by several configuration of wheel locomotion, are usually employed

for logistic and exploration.

2. *Drones* or *UAV (Unmanned Aerial Vehicles)*, aerial vehicles that are becoming popular for civilian applications.
3. *Legged robots*, which can be used in harsh environments and can mimic animals.

- *Humanoid robots*, which are inspired from human body, with a torso, two arms and hands with fingers. Their structure is very complex in order to ensure a stable bipedal motion and can not handle high payloads.
- Others: *bio-mimetic robots*, *soft robots*, etc., which aim to mimic a wide variety of animals, can have two or more legs or wings and can be designed for different environments.

In this thesis, only UGV with indoor purposes will be taken into account.

5.1.1 Kinematics of mobile robots

In order to determine a model for mobile robot's motion, it is necessary to use a bottom-up process, starting from the contribution and the constraints of each wheel. Then, taking into account the chassis geometry, an overall motion law can be defined. It is particularly important to map the robot and its component in proper local and global *reference frames*. [1]
The robot can be considered as a rigid body with wheels in a plane, therefore its dimensionality is equal to three: we need two coordinates for its position (x, y) and one for its orientation θ to describe it. For the sake of simplicity, other internal joints and degrees of freedom are ignored. As in figure 5.1, to establish the position of the robot, we need to define:

- an arbitrary *fixed inertial reference frame* $O_I x_I y_I$, defined by the basis orthogonal axes X_I and Y_I, and its origin $O : \{X_I, Y_I\}$
- a point P on the robot's chassis as its position reference
- a *local reference system* $O_L x_L y_L$ integral with the chassis, denoted by the axes $\{X_L, Y_L\}$ with origin in P

As a consequence, the pose of a robot in the global reference system will be described by a three coordinate vector $\mathbf{q}_I$.

$$\mathbf{q}_I = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \tag{5.1}$$

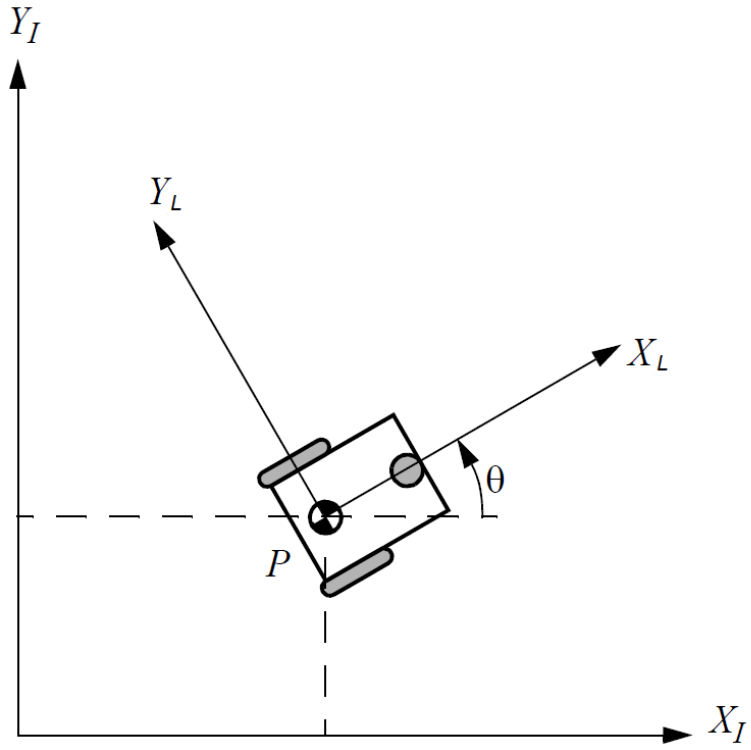


Figure 5.1: The global inertial frame $O_I x_I y_I$ and the local frame $O_L x_L y_L$ referred to the mobile robot. [1]

In order to map the motion observed in the global reference along the local one, it is necessary to apply a *transformation*: this is realized through a orthonormal transformation matrix $R(\theta)$.

$$\mathbf{q}_L = R(\theta)\mathbf{q}_I \qquad R(\theta) = \begin{bmatrix} cos\theta & sin\theta & 0 \\ -sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.2}$$

5.1.2 Kinematic model of a unicycle

A *unicycle* is a vehicle with a single steerable wheel; even though it has only a few practical applications, focusing on its kinematic can be useful to better understand the kinematic of differential mobile robots, which will be considered in this thesis. Its configuration can be described by $\mathbf{q} = \begin{bmatrix} x & y & \theta \end{bmatrix}^T$ where (x, y) are its cartesian coordinates of its contact point and θ is the orientation with respect to the x axis. It is subjected to the pure rolling constraint expressed through the following

formula (5.3), which means that in the contact point the velocity along the axis orthogonal to the motion direction is null. The line passing by the contact point along the transverse axis is called the *zero motion line*. [29]

$$\dot{x}sin\theta - \dot{y}cos\theta = \begin{bmatrix} sin\theta & -cos\theta & 0 \end{bmatrix} \dot{\mathbf{q}} = 0 \tag{5.3}$$

From (5.3) we derive that all admissible velocities are a linear combination of v and ω, which are the driving velocity and the steering velocity.

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} cos\theta \\ sin\theta \\ 0 \end{bmatrix} v + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \omega \tag{5.4}$$

5.1.3 Kinematic model of a differential drive vehicle

In the case of a differential robot, we can derive its forward kinematic model in a similar way to the unicycle. Considering a differential robot with two non steerable wheels, each with radius equal to r and with a distance from each other equal to d, a point P is taken in the middle of the two wheels, such that the distance from P to each wheel is equal to $\frac{d}{2}$. The physical inputs are the rotational velocities of the two wheels ω_l and ω_r, respectively the left and the right one. We can the find the direct correspondence between the inputs and the driving velocity v (5.5) and steering velocity ω (5.6). [28]

$$v = \frac{r(\omega_r + \omega_l)}{2} \tag{5.5}$$

$$\omega = \frac{r(\omega_r - \omega_l)}{d} \tag{5.6}$$

In another way, we could have first calculated the contribution of each wheel in the local frame $O_L x_L y_L$: considering the motion of each wheel separately, supposing the other one fixed, since P is located halfway, its instantaneous motion will have half the speed such that $\dot{x}_L = \frac{r\omega_l}{2}$ and $\dot{x}_L = \frac{r\omega_r}{2}$. In a differential robot we can simply add the contribution to obtain the motion in the local reference system. The value of $\dot{y}_L$ is necessary zero, none of the contribution can make the robot vehicle move sideways. Finally, the angular motion along the local vertical axis is considered: the right wheel spinning forward will contribute with a counterclockwise rotation $\omega_L = \frac{r\omega_l}{d}$, while the other one with a clockwise rotation $\omega_L = -\frac{r\omega_r}{d}$. If one of the two wheels moves alone, the robot will be pivoting around the contact point of the fixed wheel: the resultant motion happens around an arc with a d

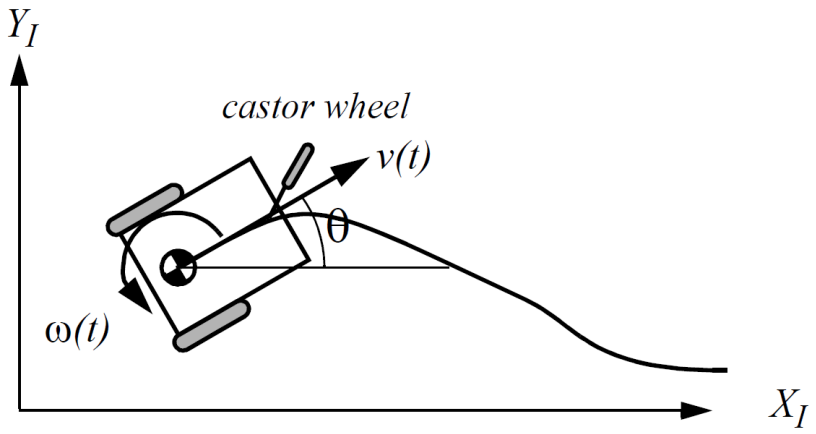


Figure 5.2: Linear velocity $v(t)$, angular velocity $\omega(t)$ and trajectory of a differential drive robot in the plane. [1]

radius. Combining the two contribution for linear and angular velocity, it leads to the same results obtained in formulas (5.5) and (5.6).

It is possible, then, to transform the obtained velocities in the inertial frame $O_I x_I y_I$.

$$\dot{\mathbf{q}}_I = \begin{bmatrix} \dot{x_I} \\ \dot{y_I} \\ \dot{\theta_I} \end{bmatrix} = R(\theta)^{-1} \begin{bmatrix} \frac{r(\omega_r + \omega_l)}{2} \\ 0 \\ \frac{r(\omega_r - \omega_l)}{d} \end{bmatrix} \tag{5.7}$$

5.2 Sensors

5.2.1 Intel Realsense tracking camera T265

The *Intel Realsense tracking camera T265* has been used to measure the ground truth position for the machine learning algorithm. The principle of working of this tracking camera consists in using the inputs from dual fisheye cameras and IMU to output its six degrees of freedom position, processing the data on a VPU processing ASIC.

Figure 5.3: Intel Realsense tracking camera T265. [30]

Component	Description
BMI055 IMU	accelerometer and gyroscope
OV9282 Fisheye Camera	monochrome wide wiew image sensor
Movidius MA215x	VPU processing ASIC
Stiffener	reinforcement housing
other components	IR cut filter, voltage regulators, etc.

Table 5.1: Component description of Intel Realsense T265 [30]

Parameter	Properties
Degrees of freedom	6
Acceleration range	±4g
Accelerometer sample rate	62.5 Hz
Gyroscope range	±2000 deg/s
Gyroscope sample rate	200 Hz

Table 5.2: Inertial measurement specifications of T265 sensor [30]

Parameter	Camera sensor properties
Active pixels	848 × 800
Sensor aspect ratio	1.06
Format	8 bit, 10-bit raw
Filter type	IR cut filter
Focus	fixed
Shutter type	global shutter
Signal interference	MIPI CSI-2, 2 X lanes

Table 5.3: T265's Fisheye image sensor properties [30]

5.2.2 Leica Laser Tracker AT403

The *Leica Laser Tracker AT403* is a portable coordinate-measuring machine, able to guarantee high precision. At the beginning of this work, it was taken into consideration to collect the ground-truth data but was later discarded due to the difficulty of communication with the ROS environment and synchronization issues. Leica AT403 registers its data on its software *Leica Tracker Pilot*, converting them into a text log file at the end of a measurement session, referring them to a chosen reference frame. Moreover when the target is moving the sampling may be not constant. Nonetheless, at the beginning of the work, a calibration of the ground-truth data has been carried out, measuring the same data using the Intel Realsense Tracking Camera T265 and the Leica Laser Tracker and calibrating the data output from the camera. The prism used to measure the position of the robot was placed on the top of the Jackal UGV.

Figure 5.4: Leica Laser Tracker AT403[31]

Parameter	**Properties**
Sensor dimensions	290 × 221 × 188 mm
Weight	7.3 Kg
Horizontal rotation	±360°
Vertical rotation	±145°
Dinamic measurement speed	10 Hz
Measurement volume	320 m
Operative temperature	from 0°C to 40 °C
Laser class	2
Resolution	0.1 μm
Accuracy	± 15 μm + 6μm/m in SMR mode ± 200 μm in B-probe mode

Table 5.4: Leica Laser Tracker AT403 technical specifications [32]

5.2.3 DecaWave TREK1000

Decawave TREK1000 is an UWB positioning evaluation kit, which includes four Decawave EVB1000 boards, which can be used both as an anchor or as a tag. Each EVB1000 is equipped with Decawave1000 IEEE802.15.4-2011 UWB wireles transceiver integrated circuit, a STM32 ARM Cortex M3 processor and a LCD display and an antenna. The TREK1000 systems uses a TWE Time of Flight measuring techniques, and it reaches an accuracy of ±10 cm in LOS conditions, while the minimum guaranteed accuracy in LOS conditions, is equal to ±30 cm for a moving tag. A proprietary software provided by Decawave allows to save data about the rangings, while positioning methods should be implemented with ad-hoc solutions. Further specifications can be found in table 5.5.
In our work four EVB1000 were used as anchor, in fixed position in the four corners of the room and a fifth was used as tag, placed on the mobile robot.

Figure 5.5: Integrated Circuit EVB1000 and the external antenna.

Parameter	Properties
Operating band	frequency bands from 3.5 GHz to 6.5 GHz U.S. FCC compliant
Center frequency	Ch. 2: 3993.6 MHz Ch. 5: 6489.6 MHz
Max. power spectral density	-41.3 dBm/MHz
Tx power	-14 dBm / -10 dBm
Tx power density	-14dBm/MHz
Preamble lenght	64 μs to 4 ms
Board dimensions	120 $\times$ 70 mm
Weight	39 g
Antenna	WB002 Omni-directional planar antenna
Ranging techniques	Pulsed TWR (Two Way Ranging)
Positioning update rate	5 Hz
Ranging precision	$\pm$10 cm

Table 5.5: DecaWave TREK1000 technical specifications [33]

5.3 Jackal UGV

In this work, the *Clearpath Jackal UGV* four wheels mobile robot was used to collect data. It is a small robot, suitable for indoor applications, and didactic robotics. It is integrated with an onboard computer, with a dual-core Intel i3-4330TE chip that runs at 2.4 GHz and 4 Gb of RAM, GPS, IMU and WiFi adapter. Fully integrated with ROS Kinetic and compatible with many accessories. It is made with an aluminium chassis and high torque 4x4 drivetrain.

For the sake of collecting data for our application, a DecaWave EVB1000 UWB transceiver has been placed in the geometrical center of the upper platform and a Intel Realsense T265 has been placed in the front of the vehicle as in the picture 5.10a and 5.10b. The detailed mechanic, hardware and software specifications are summarized in the table 5.6.

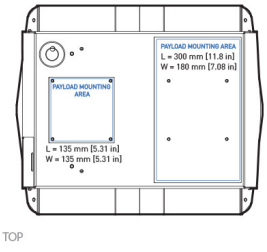


Figure 5.6: Top view

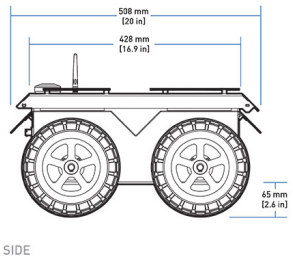


Figure 5.7: Side view

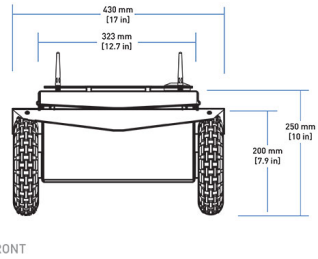
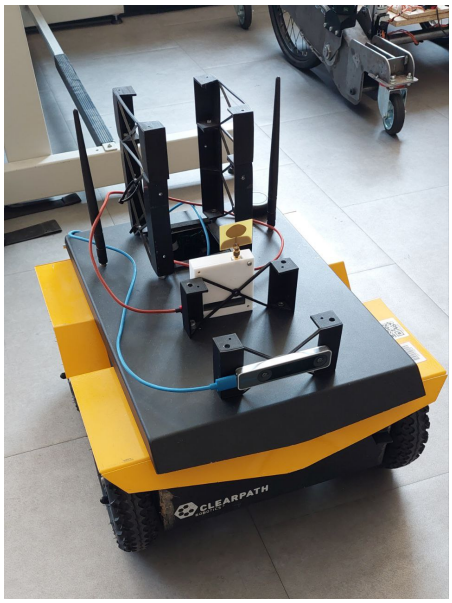
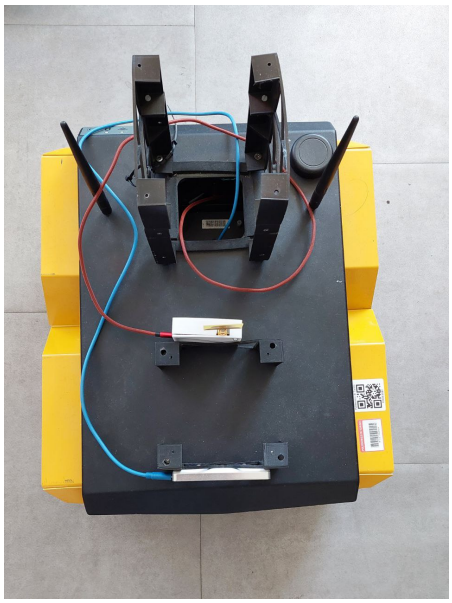


Figure 5.8: Front view

Figure 5.9: Technical specification of the Jackal UGV. [34]



(a) Side view



(b) Top view

Figure 5.10: Arrangement of the T265 camera and UWB IC on the mobile robot used for experiments.

Size and weight	
External dimensions	$508 \times 430 \times 250\ mm$
Internal storage dimensions	$250 \times 100 \times 85\ mm$
Weight	$17\ kg$
Ground clearance	$65\ mm$
Speed and performance	
Max. payload	$20\ kg$
Max. speed	$2.0\ m/s$
Drive power	$500\ W$
Battery and power system	
Battery chemistry	Lithium ion
Capacity	250 Watt hours
Charge time	4 hours
Run time	heavy usage: 2 hours
	basic usage: 8 hours
Interfacing and communication	
Control modes	Velocity, angular velocity
	Voltage
	Wheel Velocity Commands
Feedback	Battery and motor current
	Wheel velocity and travel
	Integrates GPS receiver
	Integrate gyroscope and accelerometer
Communication	Ethernet, USB 3.0, RS232
Drivers and APIs	Packaged with ROS Kinetic
Supported OS	Ubuntu, Windows
Integrated accessories	Wireless Game controller
	GPS
	IMU
	In-Board Computer
	WIFI Adapter
	Accessory Mounting Plates

Table 5.6: Technical specifications of Jackal UGV [34]

5.4 Software platforms

The software development tools used during the development of this work are described in this section. All the software ran on a machine with the *Ubuntu*

operative system, in particular the 20.04 (LTS) *Focal Fossa* version. For the real-time data collection from the Jackal UGV, the second version of ROS (*Robotic Operating System*), that is ROS2; in particular the *ROS2 Foxy* distribution was used. The motivation behind the choice of using ROS2 is that some packages developed at PiC4SeR which compute the UWB Extended Kalman Filter odometry were developed using this version. However, the Jackal UGV employs the first ROS version to publish its topics, so it was necessary to use the *rosbridge_suite* package to convert all the topics to ROS2. The code for the development of the Machine Learning part of the thesis has been written in *Python*, a high-level programming language, extensively used for its flexibility.

5.4.1 ROS

The *Robotic Operative Systems* (ROS) consists of a set of libraries designed to develop robotic applications; it is free and open source. It is a middleware, basically a set of tools that lies between the hardware and the real software, to abstract the hardware layer and manage packages. Despite the low latency, ROS is not a real-time operating system, even though it can be integrated with real-time modules. Its libraries and development tools are independent of the programming language: C++ and Python can be used.

It is based on anonymous *publishers/subscribers* communication, which allows the passage of information among the various processes. Looking at the big picture, the core concept behind its functioning is the ROS graph, a network of nodes and their connections processing data one at a time. Each node is a stand-alone. All the ROS graph is controlled by a process called ROS Master, which sets up all the communication through topics and services between nodes and controls the parameter server updates. Effective communication, which involves messages and service calls, does not pass directly through the ROS master, it sets up peer-to-peer communication. The adoption of decentralized architecture allows the nodes to be stand-alone processes, with the possibility of running on different hardware platforms, allowing off-board heavy calculations.

Information can be exchanged in different ways, among all, the principal ones are topics and services. Topics are buses, which are characterised by a unique namespace. The communication on them relies on the publisher/subscriber stream mechanism: the publisher node continuously sends data on a specific topic, while the subscriber, a node that needs to access the messages published on a topic, needs to subscribe to it. Services are different from topics and are based on a call/response mechanism: information is provided only when requested. A client node requests an action, that is served from a server node. [35]

5.4.2 Python machine learning tools

TensorFlow 2

Tensorflow is an end-to-end open-source library for machine learning which provides optimized modules for designing and implementing algorithms, initially developed by Google in 2015. It is mainly focused on training and inference for deep learning. Tensorflow consists of a set of tools implemented to provide a full pipeline to develop deep neural networks that can run on different CPUs and GPUs. Its main features include AutoDifferentiation (computing the gradients), eager execution (so that the code can be analyzed runtime), distributed computation among different devices, a set of loss functions, metrics and optimizers and a module to build neural networks. In this work, TensorFlow Keras was used to implement machine learning models.

Keras

Keras is a high-level deep learning API that runs on top of the TensorFlow 2 library, which is used as the backend. It was designed to offer the user simplicity of use, flexibility and powerful performance. It provides the high scalability and cross-platform capabilities of Tensorflow 2, as the model can be trained on many computing units such as a cluster of GPUs or TPUs, and later exported to other devices.

Chapter 6

Machine learning for localization

This chapter aims to describe thoroughly the experimental process conducted to validate the methods proposed in this thesis, that is using deep learning applied to temporal sequences of input to mitigate the odometric error in mobile robots. Two different kinds of sensors are used as inputs of the neural network, namely proprioceptive sensors, such as the wheel encoders and inertial measurement unit, and wireless sensors, such as ultra-wideband anchors and tags. A mobile robot has been teleoperated to record data, and the obtained dataset was used to train a NN model, which was tested afterwards and compared to other traditional methods.

The rest of the chapter contains a detailed description of the dataset creation, the model selection process and the achieved results.

6.1 Dataset creation

The first step of the experimental process consists of the creation of a *dataset* which will be used afterwards to train the NN model and to validate the obtained results. Even though the objective of the dataset creation is to make two different collections of data, respectively one for proprioceptive sensors and one for UWB, all the data flow was recorded at once and split afterwards.

6.1.1 Measurement environment and instrumentation

The dataset was recorded in the PIC4SeR (Polito Interdipartimental Center for Service Robotics) laboratory, which is a $10.20 \times 7.50\ m$ room, teleoperating the Clearpath Jackal four-wheeled robot around with different and irregular trajectories.

For the sake of simplicity, a fixed starting point has been established as in figure 6.1, which coincides with the fixed inertial reference frame we considered for our measurements. At the beginning of every recording run, the geometric centre of the robot has been placed in this spot, with the same orientation pointing forward, along the x-axis.

anchor	x	y	z
0	-1.712 m	1.745 m	2.463 m
1	8.063 m	1.449 m	2.469 m
2	8.110 m	-2.654 m	2.492 m
3	-1.119 m	-3.296 m	2.522 m

Table 6.1: Positions of the four anchors in the room referred to the established reference frame.

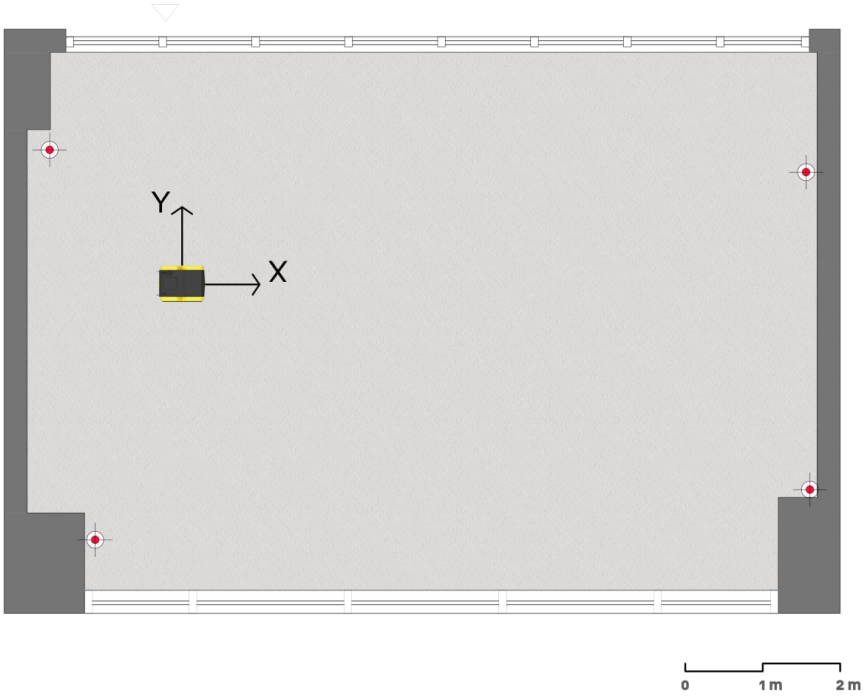


Figure 6.1: Floor plan of the lab room of PIC4SeR. The reference represents the established global inertial frame, while the four red dots represent the fixed UWB anchors.

We used four UWB anchors, placed in the four corners of the room, and the detailed positions are available in the table 6.1, with respect to the reference frame.

The main part of the room, where the robot has been driven the most, is in Line of Sight (LOS) with the four anchors as in figure 6.1.

In order to cover all the possible situations when a differential skid-drive four-wheeled robot loses the most accuracy, the driving of the vehicle included hard brakings, tight curves, sharp acceleration and turning on itself. In addition, the linear and angular speed have been limited according to table 6.2. Moreover, even though the central space, where the most measurements were taken is totally in Line of Sight, during the measurement session some people were moving in the lab, introducing some disturbance element; moreover, a table football was present in different positions, introducing more variations in the measurements. The test dataset, on the contrary, contains several traces obtained in controlled conditions, with specific trajectories aimed at testing a different kinds of conditions and with controlled disturbance elements that will be explained later on.

speed	limit
linear	0.4 m/s
angular	0.2 m/s

Table 6.2: Speed limitations imposed to the mobile robot.

The geometric centre of the Jackal mobile robot is considered as the origin of the local mobile frame attached to the robot, considering the x-axis along the travel direction, the z-axis along the vertical direction pointing up, and the y-axis completing the frame according to the *right-hand* rule. The UWB tag has been placed in the geometric centre of the Jackal robot, while the Intel Realsense T265 tracking camera is placed in the front, at 20 cm from the centre pointing towards the travel direction as in figures 5.10a and 5.10b. To obtain the measurement of the visual inertial odometry referred to the local frame a transformation (6.1) has been applied to the data.

$$\begin{bmatrix} x \\ y \\ \theta \end{bmatrix} = \begin{bmatrix} cos\theta_{t265} & -sin\theta_{t265} & x_{t265} \\ sin\theta_{t265} & cos\theta_{t265} & y_{t265} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -\ell \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -\ell cos\theta_{t265} + x_{t265} \\ -\ell sin\theta_{t265} + y_{t265} \\ 1 \end{bmatrix} \qquad \ell = 20cm \tag{6.1}$$

6.1.2 Data acquisition

The data acquisition process took place in different sessions, teleoperating the robot around the room in several runs, each one lasting several minutes. The acquisition of the data has been carried out by recording the ROS2 topics published. Since

the Jackal board computer has installed the first version of ROS and the program used to compute the odometry and UWB localization using the Extended Kalman Filter, it was necessary to use the *rosbridge_suite* to establish communication between the two versions.

The following topics have been used to carry out this work:

- */imu/data_raw*: contains the raw data measured from the IMU, such as the linear velocities and linear accelerations along the three axes.
- */joint_states*: contains the position, measured in meters, and the velocity, measured in m/s, of each one of the four wheels.
- */encoder/odometry* (called */jackal_velocity_controller/odom* in the test set): contains the odometry obtained from the encoder data evaluated through an Extended Kalman Filter by the Jackal itself. Precisely, only the position on the bi-dimensional plane (x, y) and the orientation were used.
- */odom*: contains the visual inertial odometric position obtained from the Intel Realsense t265 camera. As in the encoder odometry, only the position on the plane and the orientation along the z-axis, which is given by a quaternion, is considered. The data is transformed from the t265 position to the local reference frame and published on the */odom/base_link* topic.
- */uwb_ranging*: contains the ranging distance, the first peak power and the received power from each one of the four anchors.
- */uwb_ekf_position*: contains the estimated position obtained filtering through an EKF the ranging data of the anchors. It will be used later as a comparison of the obtained results.

The recording is done on *SQL database* file, which is later parsed using a Python script and the content of each topic is then subdivided into *comma-separated values* (CSV) files. So, for each run, several CSV files are obtained, each one containing as columns the different data and as rows the single data acquisitions with their *UNIX-16* timestamp (that is the time past from the midnight of the 1^{st} of January 1970, measured in nanoseconds).

6.1.3 Data synchronization

The final objective of the dataset creation consists in making two different groups of files with all the necessary topics to train and compare the NN models for both

topic	frequency	data	description
/imu/data_raw	30 Hz	angular_velocity	x, y, z
		linear_acceleration	x, y, z
/joint_states	15 Hz	position	4 wheels position (4)
		velocity	4 wheels velocity (4)
/encoder/odometry	15 Hz	position	x, y
		orientation	quaternion (w, z)
/odom/base_link	50 Hz	position	x, y
		orientation	quaternion (w, z)
/uwb_ranging	5 Hz	range_mes	ranging distances (4)
		first_peak_pwr	first peak power (4)
		rx_pwr	received power (4)
/uwb_ekf_position	5 Hz	position	x, y

Table 6.3: Summary of the collected topics.

situations, namely the UWB localization and the wheel odometry.

The first problem that emerged during a check of the recorded data is that the T265 odometric data suffers from a non-calibration of the scale: the position reported along a trajectory appears to be off by 20-30 % of the real scale. The trace of the trajectory, when plotted on a bi-dimensional plane, appears to be proportional to the real ground truth value, collected by the Leica laser tracker AT 403, but on a different scale for each trace. This problem was solved with subsequent data post-processing. During the trajectory, the mobile robot is placed along a line parallel to the starting line, placed 5 m in front of the starting point. During post-processing this was used to calibrate the trajectory: the scale factor was obtained by multiplying the (x, y) position by a *scale factor*.

$$d_{real} = \alpha \cdot d_{measured} \qquad \alpha = \frac{ref_{real}}{ref_{measured}} \tag{6.2}$$

The second problem that occurred is the synchronization of the ground truth obtained from the Laser Tracker Leica AT403: in contrast to all the other data, the obtained position does not have a *UNIX-16* timestamp and they are not published on a ROS topic. Moreover, even if the position is extremely accurate, the sampling frequency is not constant: it reaches a maximum of 10 Hz when the robot does not move but is highly irregular during motion. The fact that the resolution of the time is equal to 1 s and the frequency variability does not make it suitable for the synchronization of other data for the learning process. However, superposing the

traces of the corrected visual inertial odometry and the sequence of the measured points by the laser tracker, shows how the VIO is extremely precise, namely with a 3 cm tolerance, and can be used as ground truth in the learning process.

A third problem that arose during a first check on the data recorded by Intel Realsense T265: in case of a sudden change of light intensity or the presence of someone walking in front of the camera, an instantaneous shift along the horizontal plane, perpendicularly to the travel direction occurs. The entity of these shifts can vary, from a few centimetres to several tens of centimetres. This problem, which affects the smoothness of the trace and compromises the learning process is addressed in two ways. The first solution consists in *smoothing* the trace by averaging it along 25 measured points, which correspond to about 0.5 s: it helps to flatten the smaller shifts, namely the ones smaller than 10 cm, without compromising the trace. The second solution, adapted for the shifts larger than 10 cm, is to cut the trace of the visual-inertial odometry, preventing the compromised data to spoil the learning process.

To solve the issues related to the visual inertial odometry, a Python script has been written. First of all, the value of the three coordinates of the robot, the position x, y and the orientation theta, are plotted along the time. Then the time interval in which the robot has been placed on the 5 m line is selected, and the value of the position is averaged and used to obtain a scale factor to rescale the entire trace. By doing so, the trajectory with the correct proportion is obtained. Afterwards, the trajectory is plotted on an $x - y$ plane and the aforementioned shifts are shown together with their absolute value. Later on, the trajectory averaged along 25, 35, and 50 data points are shown, to display the improvement of the smoothing. If necessary, the trace can be cut at an optional point. In this work, it was decided to cut the traces in case of a shift larger than 10 cm and to average 25 points.

Finally, the real synchronization process takes place: the objective of this place is to obtain two sets of comma-separated values files, namely one for encoder odometry and one for UWB localization, to load directly onto the training and testing program.
For the wheel odometry, the topics used were:

- /imu/data_raw
- /joint_states
- /encoder/odometry
- /odom/base_link

Since it can be seen, by analyzing the timestamps of the data, that even if IMU and encoder topic messages display a constant frequency, the time interval between arrival time is largely irregular. Therefore, the data values were interpolated by a linear spline and sampled at 10 Hz.
In the case of UWB localization, the considered topics were:

- /uwb_ranging
- /uwb_ekf_position
- /odom/base_link

In contrast to the encoder odometry case, here the arrival time of the topics of UWB ranging and the respective position obtained through EKF is regular, about 5 Hz, since no bridge between ROS and ROS2, and the second one is evaluated right after the receiving of the first one. Therefore, the time of the EKF position is used as a deadline for synchronization: the last message sent on the other topics before the deadline is taken into consideration for the synchronization purpose.

In the following table 6.4, the properties of the two obtained datasets are summarized: these datasets were used to train the two models of neural networks and to compare the results in the following sections of this chapter.

	UWB	encoder
n. traces	35	36
n. samples	$28.09 \cdot 10^3$	$62.15 \cdot 10^3$
time	$5.62 \cdot 10^3$ s	$6.23 \cdot 10^3$ s

Table 6.4: Summarizing table of the datasets' properties

6.2 Neural Network model selection

After the creation of the dataset, the following stages in the development of this work consisted in creating a training environment and carrying out a *gridsearch* over a set of hyperparameters, to identify the best ones for each application. It consists of training a network with all the combination of predefined values of parameters, and establish the best considering the minimum value achieved by the loss function or other metrics. These activies were developed using Jupyter Notebook, with a GPU running in the PIC4SeR laboratory. This section describes in detail the methodological procedure.

6.2.1 Training procedures

As a first step, the datasets are split up into overlapping time sequences of n_{points} elements, by shifting a time window one-time step per time. So it is possible to model the problem as if the output y_t at time t, does not depend only on the input x_t but also on the previous $x_{t-1}, \ldots, x_{t-n_{points}}$ inputs. Thus, the NN model will be fed as input with a time sequence, the received output will be related to the last time instant and the final trajectory is then reconstructed at the end, losing just the n_{points} initial instants.

In the case of the encoder-inertial odometry dataset, we took into consideration that all the measurements accomplished by proprioceptive sensors are related to the local frame of the mobile robot and can not be reconducted to a global reference frame of the room. Therefore, a transformation was applied to the robot trajectory: at each time step, the absolute position and orientation were transformed to the movement, in terms of spatial shift and angle, that were done with respect to the previous timestep, with respect to the local frame of reference. This procedure allowed us to easily predict the motion of each time step and reconstruct the whole trajectory, or part of it, afterwards. In the case of UWB localization, instead, we considered the global position in the room. Moreover, in order to speed up the training, all the data were normalized with a standard normalization, i. e. with mean $\mu = 0$ and standard deviation $\sigma = 1$.

The same kind of grid search has been carried out for both datasets, but different results were obtained: preliminary work was carried out to identify the neural network type that suits best each problem. It emerged that, if in the case of the UWB dataset, good results can be achieved using both a feedforward neural network or a LSTM neural network, in the case of encoder odometry it is necessary to adopt an LSTM architecture since the multilayer perceptron does not produce remarkable results. So, we proceeded to run a grid search with both datasets, taking into account a set of hyperparameters suitable for inference on resource-constrained applications, that could be the board CPU of a mobile robot. All the parameters and the best models will be illustrated in detail in the following sections.

6.2.2 Model selection for encoder-inertial odometry

The model selection process for encoder-inertial odometry started by identifying the model of the neural network and the possible sets of values for the hyperparameters. At first, some experiments with a dense neural network were carried out, but no significant results emerged, so an LSTM architecture was tried with considerable outcomes. The length of the input sequence, the values of the neurons of each

layer, and the number of the hidden layer itself have been selected taking into account the training time, which is significantly longer for deep networks or long sequences. Moreover, the total number of parameters, which is defined by the number of hidden layers and the number of neurons, needs to be kept low to make the network run on constrained-resources platforms. The hyperparameter selected for the grid search is reported in detail in table 6.5.

parameter	values set
# hidden LSTM layers	{1, 2, 3, 4}
# neurons	{16, 32, 64, 96}
learning rate	{0.001, 0.0001}
n_{points}	{1, 2, 5, 10, 20}

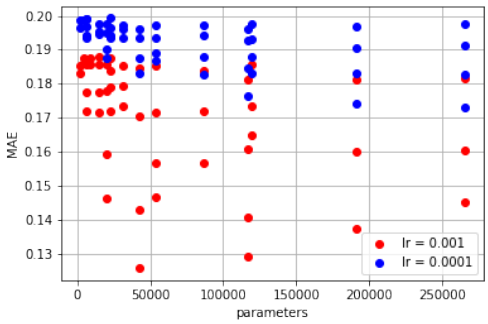
Table 6.5: Values of hyperparameters tried during the grid search on the encoder-inertial odometry

We also chose to adopt the mean absolute error as an error function, defined in Chapter 4 as 4.19, due to its robustness to outliers points and better performance in a preliminary phase. Moreover, we adopted the ADAM optimizer which according to [25] is computationally efficient and has little memory requirement. LSTM neurons are chosen to have a sigmoid recurrent activation function and a ReLU output activation function while we chose a dense output layer with a linear activation function. All the neural networks were trained for 50 epochs.

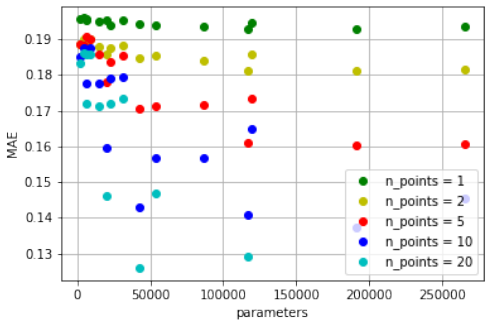
From the grid search results analysis, it is possible to see at first sight 6.2a that a learning rate $\eta = 0.001$ generally performs better than the other values: so we'll focus shortly on the performances and the complexity of the models with this feature. It can be seen 6.2b that the models trained with longer input sequences perform better, being able to capture the evolution of the motion over time. Also it can be seen in figure 6.2c and 6.2d that, a larger number of neurons seems to provide better results, while the opposite can be said with the number of hidden layers: this allows to keep the number of parameters bounded. Finally, the chosen model has one LSTM hidden layer with 96 neurons, a learning rate equal to 0.001 and a 20 timesteps-long input sequence.

6.2.3 Model selection for UWB localization

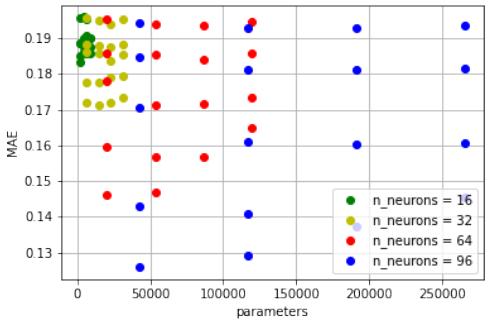
In contrast to the encoder odometry neural network model, during a preliminary phase it emerged how for UWB localization, in the context of this work, a fully connected neural network performs as well as LSTM architecture. Therefore the two kinds of architecture will be considered and tested afterwards.



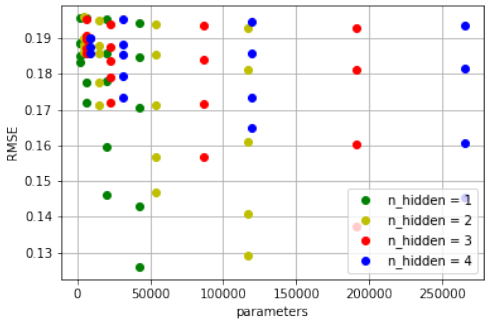
(a) Different values of learning rate.



(b) Different sequence lenght, n_{points}



(c) Different values of $n_{neurons}$



(d) Different number of hidden layers n_ℓ

Figure 6.2: Analysis of the results of the grid search for encoder-inertial odometry LSTM neural network. The x-axis indicates the number of parameters, the y-axis the Mean Absolute Error (MAE). Figure 6.2b, 6.2c and 6.2d refer to the value of learning rate $\eta = 0.001$ only.

parameter	values set
# hidden dense layers	{1, 2, 3, 4}
# neurons	{32, 64, 128, 256}
learning rate	{0.1, 0.0, 0.001, 0.0001}
n_{points}	{1, 2, 5, 10, 20, 50, 100}

Table 6.6: Values of hyperparameters tried during the grid search for UWB localization on a FFNN.

First of all we consider a classical FFNN architecture. The explored hyperparameter space is defined with the same criteria as the other case: for every layer, a ReLU activation function is chosen, due to its approximation properties. The Mean Squared Error (MSE) , defined in equation 4.18, is chosen due to the higher

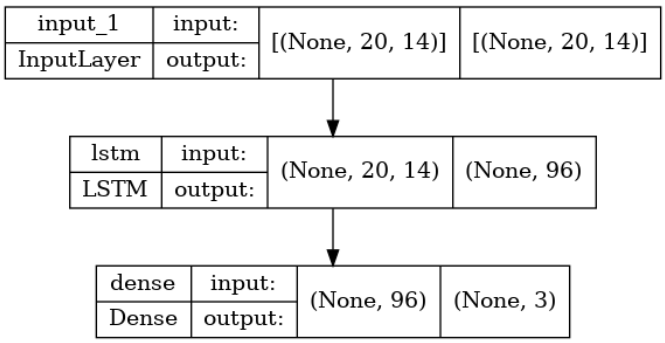


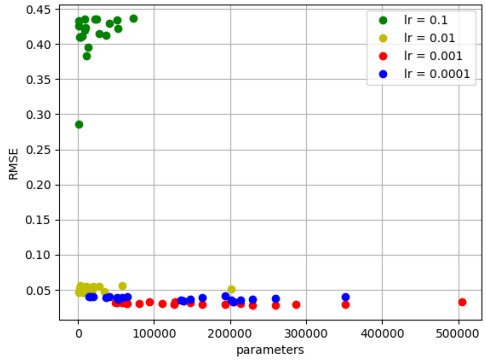
Figure 6.3: The chosen neural network model for encoder-inertial odometry. It has one LSTM layer with 96 neurons and a dense output layer. The input sequence is 20 time steps long

weight that the farther points are given; the ADAM optimizer is used and all the models are trained for 50 epochs.

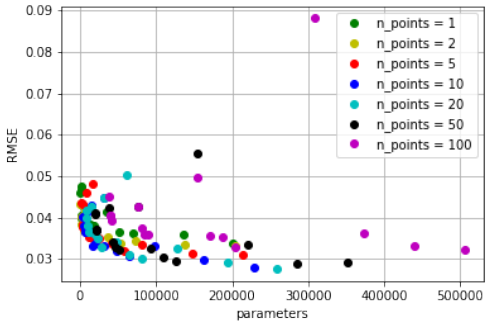
As in the previous case, we can see in 6.4a that a learning rate $\eta = 0.001$ leads to better performance, while higher values do not allow a good convergence of the model: we'll analyze more in detail the models with this feature. From 6.4b it emerges how input sequences with a length between 10 and 50 perform better than longer or shorter sequences: shorter sequences have few parameters but do not reach perform as well as the aforementioned ones, while longer sequence models are characterized by many parameters and do not perform as well. Moreover, it can be seen from figures 6.4c and 6.4d respectively, that a higher number of neurons or hidden layers are characterized by a higher number of parameters but manage to achieve better performances. Finally, a four layers model with 256 neurons each was chosen, with an input sequence which is 20 timesteps long as in figure 6.5.

Later, we focused on the LSTM architecture. In this case, we consider the same parameter space as the LSTM network for encoder odometry. The only difference is that only a learning rate $\eta = 0.001$ was tried, after considering the results of the previous grid search processes. All the hyperparameter sets can be found in table 6.7.

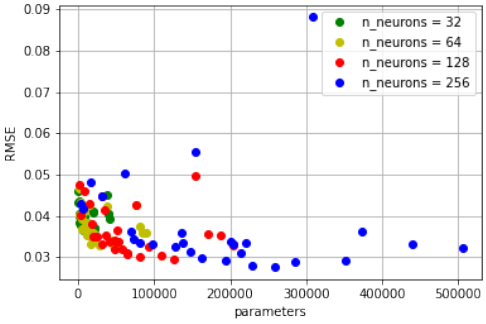
From a qualitative analysis of the graphs in figures 6.6a, 6.6b, 6.6c and 6.6d, the same considerations of the previous cases can be done. In particular, it can be seen clearly how a higher number of points provides better performance. Also, the number of neurons influences the accuracy of the result, in fact, more neurons make the network model achieve a smaller value of the loss function. The number of layers, instead, seems to not make a great difference in terms of the final score and keeping the other hyperparameter constant. Finally, since there is less than 0.1%



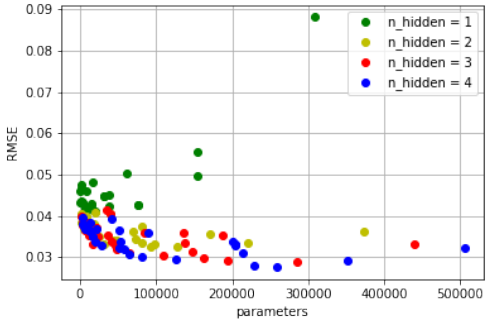
(a) Different values of learning rate.



(b) Different sequence lenght, n_{points}



(c) Different number of neurons $n_{neurons}$



(d) Different number of hidden layers n_ℓ

Figure 6.4: Analysis of the results of the grid search for UWB localization on a FFNN. The x-axis indicates the number of parameters, the y-axis the Root Mean Square Error (RMSE). Figure 6.4b, 6.4c and 6.4d refer to the value of learning rate $\eta = 0.001$ only.

parameter	values set
# hidden dense layers	{1, 2, 3, 4}
# neurons	{16, 32, 64, 96}
learning rate	{0.001}
n_{points}	{1, 2, 5, 10, 20}

Table 6.7: Values of hyperparameters tried during the grid search for UWB localization on a LSTM architecture.

of difference between the best model, the one with the fewer parameters is chosen. The chosen network consists of a three-layer composed of 96 LSTM neurons each,

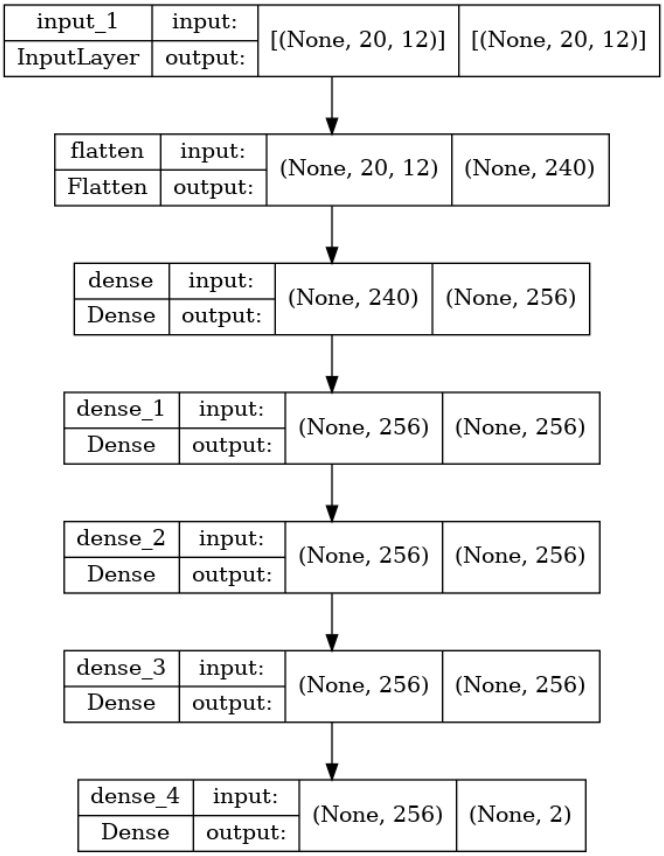
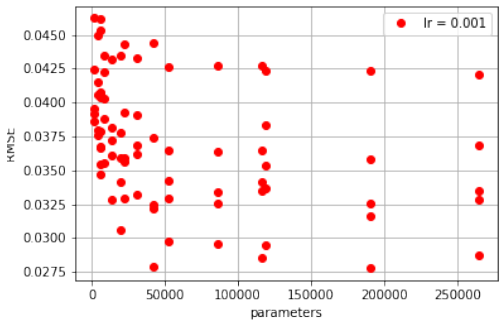


Figure 6.5: The chosen neural network model for UWB localization. It has four dense layers with 256 neurons and a dense output layer. The input sequence is 20 timesteps long.

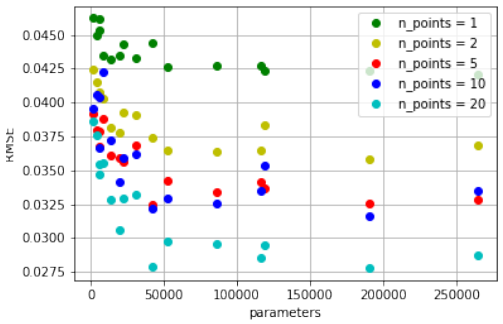
plus a dense output layer, as in figure 6.7. Moreover, the input consists of a 20-time steps sequence.

6.3 Results analysis and validation

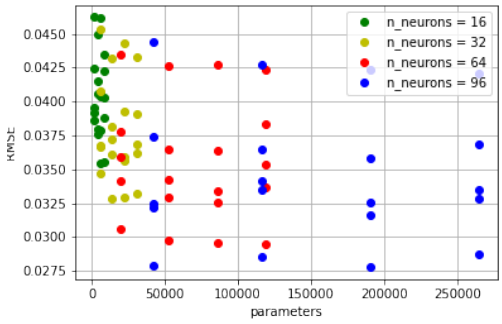
The previous parts of this chapter were devoted to collecting a training dataset, finding the best model through a grid search and training it. This part, instead, focuses on the evaluation of the proposed solution through the collection of a test dataset and the analysis of its results. Moreover, specific trajectory and scenario arrangement was employed, to better understand the behaviour of the chosen approach in different reproducible situations. The different adopted evaluation methods will be applied to the outcomes of an Extended Kalman Filter and the eventual differences or discrepancies will be considered and discussed.



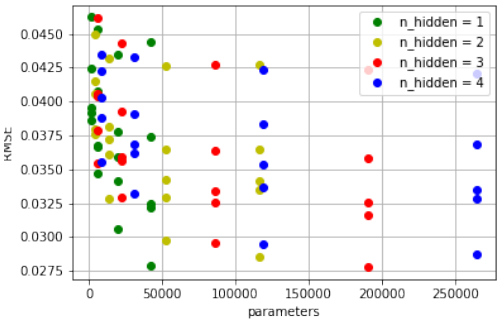
(a) Different values of learning rate.



(b) Different sequence lenght, n_{points}



(c) Different number of neurons $n_{neurons}$



(d) Different number of hidden layers n_ℓ

Figure 6.6: Analysis of the results of the grid search for UWB localization on a LSTM neural network. The x-axis indicates the number of parameters, the y-axis the Root Mean Square Error (RMSE).

6.3.1 Encoder odometry results

To evaluate the error mitigation of our model regarding the encoder odometry localization system, a test dataset has been collected. The robot was driven in the same environment and conditions as during the collection of the training dataset, with the only difference being that the trajectories were planned to highlight some specific situations. In fact, for the purpose of achieving a reproducible and as complete as possible evaluation, we considered the conditions when a differential drives mobile robot loses the most encoder odometry accuracy and teleoperated the robot along the chosen path. As we assumed in the previous chapters, the considered type of mobile robot loses accuracy in case of curves, hard-braking or strong accelerations. So the following types of trajectories were chosen:

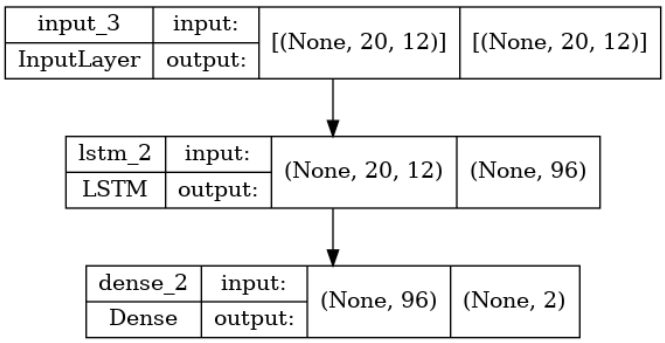


Figure 6.7: The chosen neural network LSTM model for UWB localization. It has fa single LSTM layer with 96 neurons and a dense output layer. The input sequence is 20 timesteps long.

- *Circle path*: this kind of trajectory, since the rotation of the robots happens only in a single direction, avoids the fortuitous cancellation and compensation of the error, The main component of the error is expected to be present mainly on the angle of the robots with respect to the inertial frame, due to its accumulation.
- *Infinite* ∞ *shaped path*: this trajectory consists of a series of curves in the alternate direction of rotation. This kind of trajectory may lead to casual compensation of the yaw error: if this phenomenon does not happen in a balanced way, the encoder odometry is expected to degenerate and lose its accuracy.
- *Casual path*: this path is made up of different components such as curves, brakings and acceleration. The expected result is a rapid loss of accuracy of the calculated encoder odometry.

To evaluate the performance of the model and compare it with the classical Extended Kalman Filter approach, the following metrics are used:

- *Mean Absolute Error* (MAE). In the case of the position error, the MAE is evaluated over the tracking error in the normal direction, i. e. the mean value of the length of the segment defined by the ground truth position and the estimated position at a certain time instant t. If the orientation error is considered, the MAE is evaluated by averaging the absolute value of the difference between the ground truth orientation and the estimated orientation.
- *Mean Absolute Trajectory Error* (m-ATE). In both cases, it is defined as the average of the absolute value of the error reached during a certain time of navigation. In this case, we consider the m-ATE during a period of 1 s, which

n.	lenght [s]	description
0	150	circle
1	115	circle
2	130	circle
3	110	circle
4	120	circle
5	110	circle
6	110	circle
7	150	circle
8	195	∞-shape
9	60	∞-shape
10	195	∞-shape
11	225	∞-shape
12	110	irregular
13	130	irregular
14	100	irregular
15	95	irregular
16	120	irregular
17	125	irregular
18	120	irregular

Table 6.8: List of the odometry experiment and their path characteristics.

means the average error after 10 timesteps, since the sampling frequency is fixed at 10 Hz.

- *Cumulative Absolute Trajectory Error* (c-ATE). It sums the absolute value error of every timestep up to a given point in a trajectory. This kind of error evaluation shows better the trend since is less influenced by fortuitous error compensations.

The following tables report in detail the outcome of all the conducted experiments. In particular, for each category of experiments, two tables are present: the first referred to the positioning error and the second referred to the orientation error. Each table reports a different kind of error related to the proposed NN model, the EKF and the relative improvement or worsening brought by the NN with respect to the EKF.

In all the reported graphs the black traces are referred to the ground truth data, the blue ones to the EKF estimation and the orange ones to the LSTM neural network estimation. The dashed lines, when present, represent the mean values of

the considered parameter.

Experiment 1: round trajectories

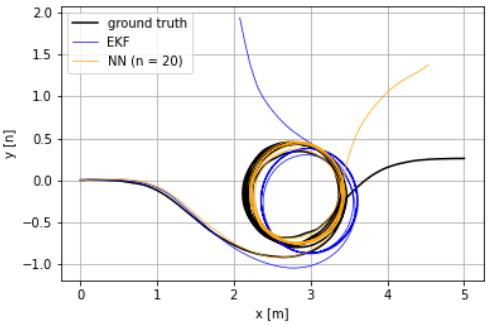
The round path experiment aims at testing the mitigation of the error since, as said above, there is no possibility of fortunate compensation. As it is possible to notice from the experiment graphs and tables, the LSTM neural network can strongly mitigate the unbounded growth of the orientation error and, as a consequence, of the positioning error. From table 6.10, it immediately stands out how the m-ATE error, which can be considered as a short-term prediction error is improved by around 70 %, which leads to an improvement of the MAE(θ) along the trajectory and the cumulative error c-ATE(θ). From table 6.9, it emerges how the position m-ATE has a smaller improvement, around 30 %, thus the strong improvement on the overall trajectory MAE, can be mainly conferred to the improvement in the angle.

n.	MAE	[m]		m-ATE	[m]		c-ATE	[m]	
	NN	EKF	$\Delta_{\%}$	NN	EKF	$\Delta_{\%}$	NN	EKF	$\Delta_{\%}$
0	0.342	1.029	66.7	0.046	0.077	39.4	4.69	7.77	39.5
1	0.192	0.526	63.5	0.028	0.039	28.1	3.01	4.31	30.1
2	0.193	1.069	81.9	0.032	0.070	54.3	3.65	7.51	51.3
3	0.240	0.691	65.2	0.022	0.033	31.4	2.31	3.33	30.6
4	0.118	0.484	75.4	0.031	0.057	45.3	2.94	5.74	48.6
5	0.570	0.548	-3.9	0.037	0.036	2.95	3.63	3.74	2.9
6	0.317	0.548	42.0	0.035	0.045	20.8	3.14	4.07	22.8
7	0.342	0.698	50.9	0.037	0.044	13.9	4.81	6.37	24.4
mean	0.289	0.746	57.4	0.034	0.051	30.1			31.9

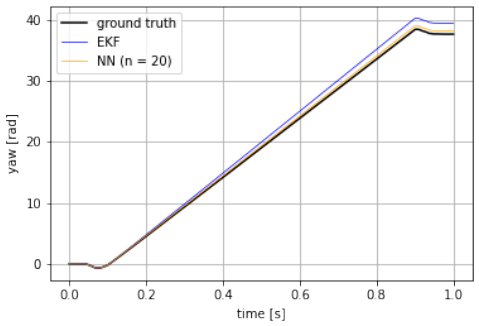
Table 6.9: Results obtained in the evaluation of the position error in round-shaped trajectory experiments. The last row refers to the overall mean.

n.	MAE	[rad]		m-ATE	[rad]		c-ATE	[rad]	
	NN	EKF	Δ%	NN	EKF	Δ%	NN	EKF	Δ%
0	0.390	1.458	73.2	0.0106	0.0370	71.4	1.23	4.88	74.7
1	0.288	0.969	70.2	0.0101	0.0365	72.3	105.3	3.76	76.2
2	0.178	1.169	84.7	0.0082	0.0362	77.0	0.97	4.30	77.2
3	0.306	0.914	66.5	0.0112	0.0339	66.8	0.88	3.13	71.8
4	0.018	0.429	95.6	0.0055	0.0172	67.6	0.94	2.98	68.1
5	0.050	0.595	91.3	0.0062	0.0234	73.4	0.95	3.00	68.2
6	0.051	0.499	89.7	0.0067	0.0206	67.1	0.86	2.83	69.4
7	0.103	0.610	83.0	0.0063	0.0188	66.5	1.27	3.93	67.4
mean	0.189	0.850	80.5	0.0080	0.0280	70.7			71.6

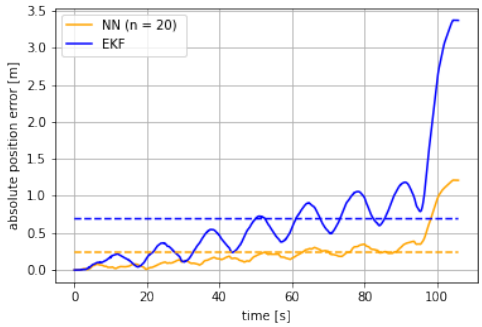
Table 6.10: Results obtained in the evaluation of the orientation error in round-shaped trajectory experiments. The last row refers to the overall mean.



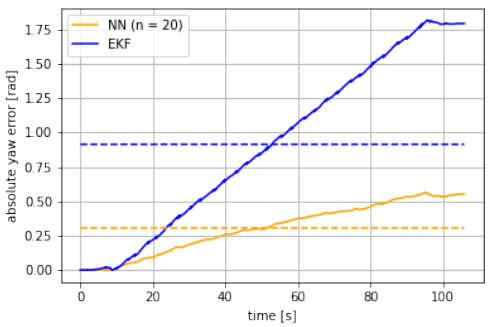
(a) Trajectory on (x, y) plane.



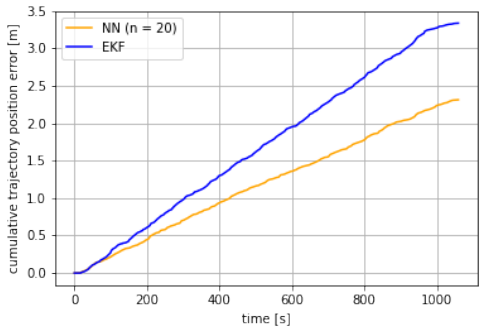
(b) Variation of θ during time.



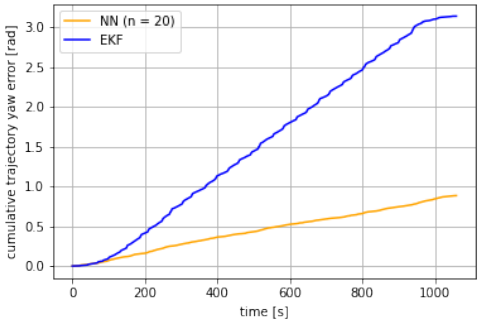
(c) Absolute position error.



(d) Absolute yaw error.

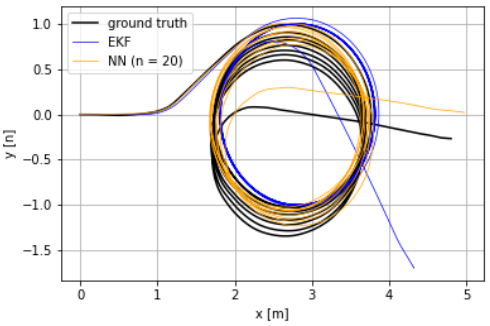


(e) Position c-ATE.

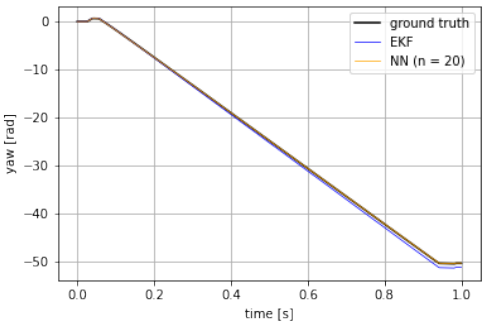


(f) Orientation c-ATE.

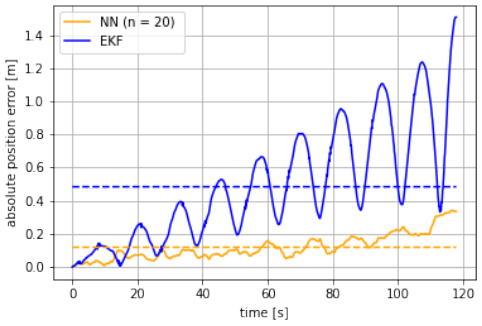
Figure 6.8: Graphs representing the ground truth and estimated trajectory of the experiment n. 3, a counter-clockwise round-shaped path. It is clearl how the LSTM-based approach strongly reduces the error on the orientation coordinate.



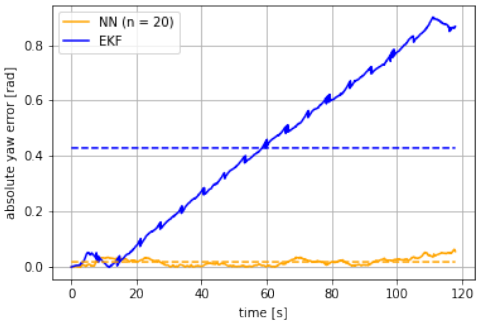
(a) Trajectory on (x, y) plane.



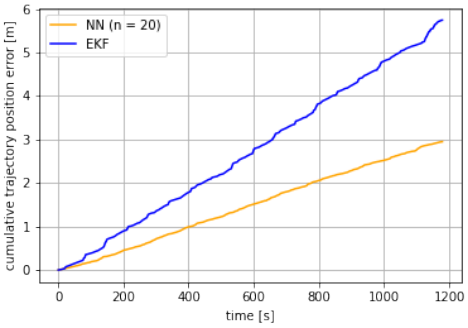
(b) Variation of θ during time.



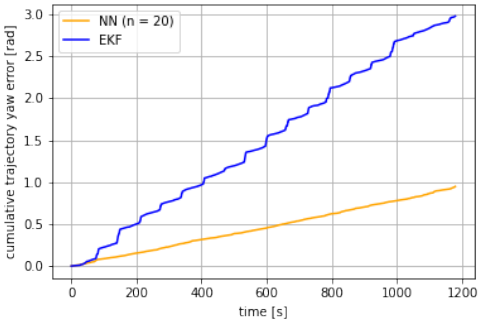
(c) Absolute position error.



(d) Absolute yaw error.



(e) Position c-ATE.



(f) Orientation c-ATE

Figure 6.9: Graphs representing the ground truth and estimated trajectory of the experiment n. 4, a clockwise round-shaped trajectory.

Experiment 2: ∞-shaped trajectories

As stated above, the ∞-shaped trajectory allows some fortunate odometry compensations, because of the consecutive curves in opposite directions. Despite that, if

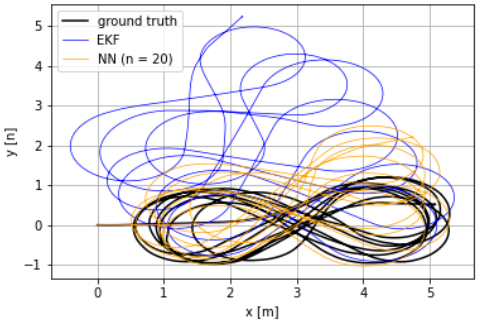
the estimation of the change in orientation is not accurate, the consequence is the total degradation of the odometry estimation, as it can be observed in figure 6.10a, in the EKF estimated path. As in the previous experiment, the NN proposed model is able to improve the m-ATE of the orientation of more than 60 %, improving the Mean Absolute Error of the orientation of more than 65 %. The improvement in terms of mean absolute position error, (MAE) are evident from table 6.11 and can also be noticed at first sight in figure 6.10a.

n.	MAE	[m]		m-ATE	[m]		c-ATE	[m]	
	NN	EKF	Δ%	NN	EKF	Δ%	NN	EKF	Δ%
8	0.459	1.446	68.2	0.028	0.041	31.3	4.80	7.30	34.1
9	0.110	0.293	62.3	0.054	0.080	32.5	1.95	2.97	34.4
10	0.429	0.916	53.1	0.030	0.040	24.7	5.22	6.83	23.5
11	0.521	1.857	71.9	0.025	0.046	45.5	5.00	9.07	44.8
mean	0.385	1.136	63.8	0.029	0.045	34.4			30.8

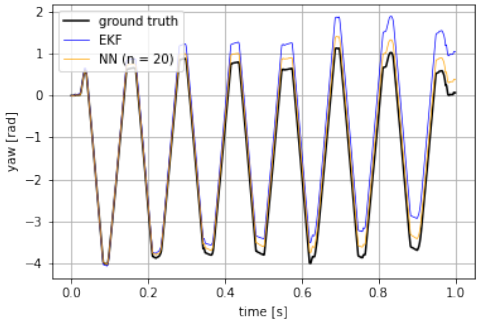
Table 6.11: Results obtained in the evaluation of the position error in ∞-shaped trajectory experiments. The last row refers to the overall mean.

n.	MAE	[rad]		m-ATE	[rad]		c-ATE	[rad]	
	NN	EKF	Δ%	NN	EKF	Δ%	NN	EKF	Δ%
8	0.095	0.523	81.7	0.0077	0.0226	65.8	1.60	4.22	62.0
9	0.077	0.186	58.6	0.0105	0.0252	58.1	0.47	1.13	58.3
10	0.151	0.373	59.5	0.0090	0.0208	56.6	1.72	4.05	57.3
11	0.183	0.459	59.9	0.0078	0.0223	64.6	1.83	4.96	63.0
mean	0.124	0.393	66.0	0.0083	0.222	62.3			60.1

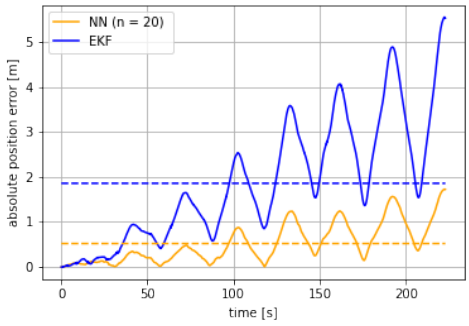
Table 6.12: Results obtained in the evaluation of the orientation error in ∞-shaped trajectory experiments. The last row refers to the overall mean.



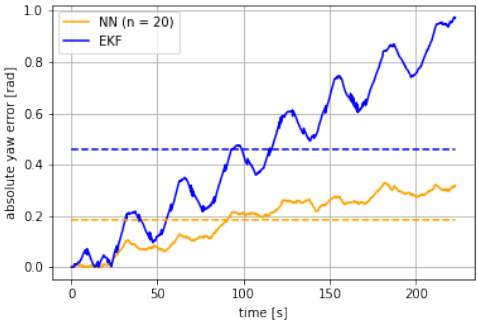
(a) Trajectory on (x, y) plane.



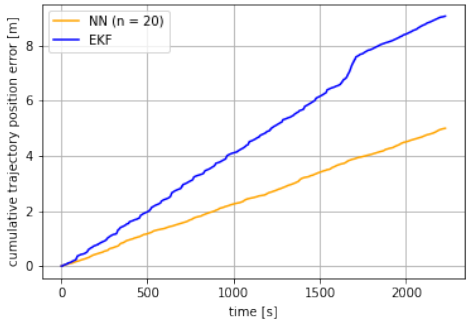
(b) Variation of θ during time.



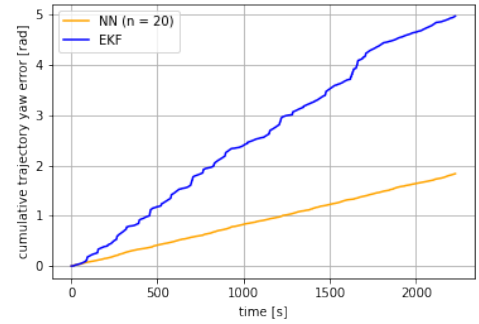
(c) Absolute position error.



(d) Absolute yaw error.



(e) Position c-ATE.



(f) Absolute orientation error.

Figure 6.10: Graphs representing the ground truth and estimated trajectory of the experiment n. 11, a ∞-shaped trajectory. From figure 6.10a it is evident how the LSTM neural network keeps the error bounded, without leading to a degradation of the odometry. From figures 6.10c and 6.10d, it shows how this kind of trajectory is prone to fortunate error compensations.

Experiment 3: irregular trajectories

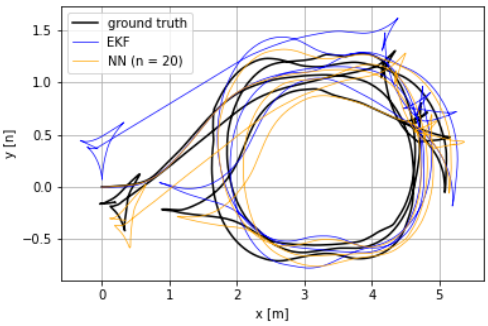
The irregular trajectory tests can be intended as a more realistic navigation test, with a non-regular path, brakings and turns on the robot itself. Even though the different kinds of conditions, the neural network model leads to the same results and improvement. The strong improvement of the short-term orientation error, and consequently the average one and the long-term one, cause an overall improvement of the position error. In general, the Mean Absolute Error is improved by 60 % with respect to the EKF outcome.

n.	MAE	[m]		m-ATE	[m]		c-ATE	[m]	
	NN	EKF	$\Delta^{\%}$	NN	EKF	$\Delta^{\%}$	NN	EKF	$\Delta^{\%}$
12	0.284	0.906	68.6	0.036	0.045	20.4	3.34	4.83	30.7
13	0.220	0.530	58.5	0.029	0.044	34.5	3.52	4.89	28.0
14	0.274	0.720	61.8	0.021	0.041	46.9	2.18	3.51	37.9
15	0.227	0.585	61.0	0.024	0.046	47.1	2.14	3.62	40.9
16	0.189	0.340	44.3	0.028	0.043	32.8	3.15	5.00	36.8
17	0.180	0.253	28.5	0.034	0.047	27.3	3.72	5.36	30.67
18	0.093	0.722	87.0	0.027	0.041	34.6	2.77	4.34	36.1
mean	0.205	0.600	59.7	0.028	0.043	34.7			34.0

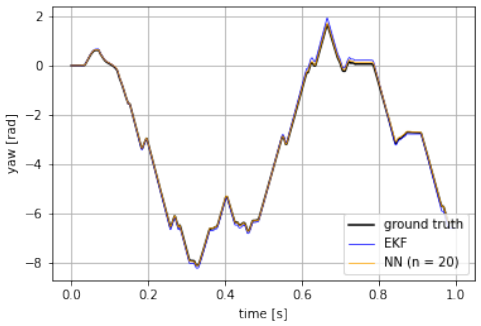
Table 6.13: Results obtained in the evaluation of the position error in irregular-shaped trajectory experiments. The last row refers to the overall mean.

n.	MAE	[rad]		m-ATE	[rad]		c-ATE	[rad]	
	NN	EKF	$\Delta\%$	NN	EKF	$\Delta\%$	NN	EKF	$\Delta\%$
12	0.170	0.611	72.0	0.0103	0.0256	59.7	1.30	3.32	60.6
13	0.144	0.186	22.6	0.0097	0.0210	53.5	1.34	3.25	58.6
14	0.159	0.435	63.4	0.0105	0.0253	58.4	1.08	2.65	59.2
15	0.134	0.403	66.5	0.0094	0.0221	57.2	1.03	2.44	57.4
16	0.104	0.198	47.2	0.0085	0.0223	61.9	1.21	3.29	63.0
17	0.028	0.091	68.5	0.0092	0.0212	56.2	1.39	3.06	54.4
18	0.039	0.498	92.1	0.0104	0.0220	52.4	1.14	2.60	55.9
mean	0.108	0.335	61.2	0.0096	0.0226	57.1			58.4

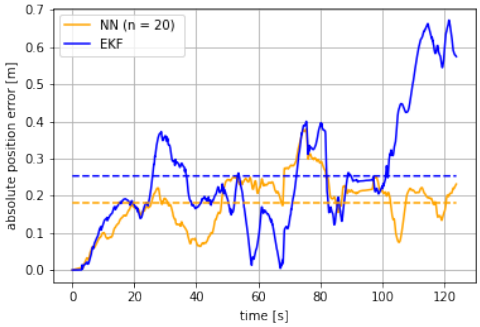
Table 6.14: Results obtained in the evaluation of the orientation error in irregular-shaped trajectory experiments. The last row refers to the overall mean.



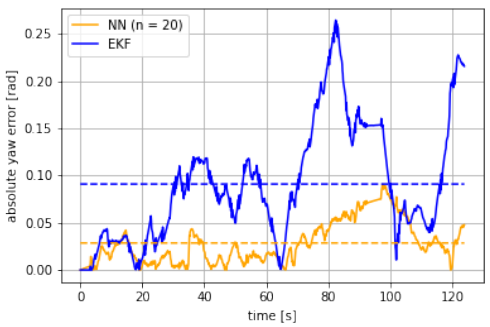
(a) Trajectory on (x, y) plane.



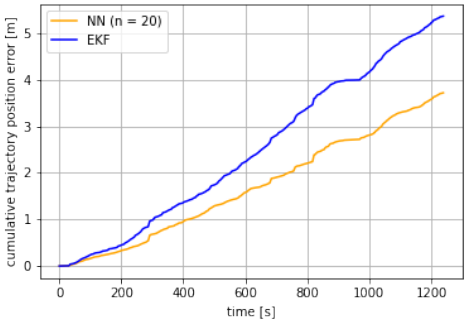
(b) Variation of θ during time.



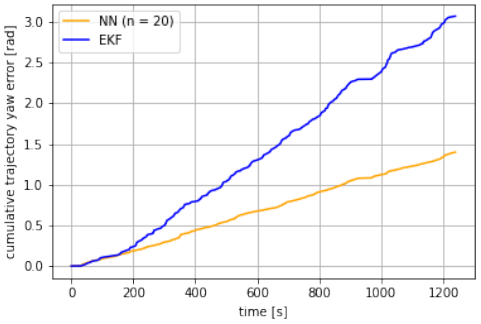
(c) Absolute position error.



(d) Absolute yaw error.



(e) Position c-ATE.



(f) Orientation c-ATE.

Figure 6.11: Graphs representing the ground truth and estimated trajectory of the experiment n. 17.

6.3.2 UWB localization results

To evaluate the error mitigation of our model regarding the UWB localization system, a test dataset has been collected. During the collection of the training

dataset, the line of sight and non-line of sight situations were not distinguished clearly, but both situations were present at the same time, with many people moving in front of the anchors and some furniture not always in the same position. On the contrary, during the collection of the test set, we tried to recreate some distinct situations in which we expect different behaviours. The following types of situations were chosen:

- *Line of Sight*: the whole trajectory does not present any obstacle in the direct line between the tag and the four anchors. In this case, we expect a performance similar to the Extended Kalman Filter outcome, since it qualitatively achieves an accurate performance.
- *Non-Line of Sight*: in this case during almost all the length of the trajectory at least one out of the four anchors is in NLOS conditions. In fact, a table football and some metal furniture have been placed in the middle of the room and the mobile robot was moved around. Moreover, some people were moving around. Wood and human bodies are known to screen the UWB wavelength, while metals act as a reflecting material. We expect qualitatively a bit of improvement with respect to the EKF outputs since the training included several NLOS situations
- *Mixed scenario*: the data were collected in the same condition as the training dataset, in a situation that can be defined as 'realistic', with people walking around. An improvement is expected since it represents the situation the NN was trained on.

n.	lenght [s]	description
0	110	Line of Sight
1	115	Line of Sight
2	110	Line of Sight
3	150	Line of Sight
4	100	Non-Line of Sight
5	80	Non-Line of Sight
6	126	Non-Line of Sight
7	255	Dynamic
8	170	Dynamic
9	265	Dynamic
10	125	Dynamic

Table 6.15: List of the UWB validation experiment and their characteristics.

To evaluate the performance of the proposed approach and compare it with the Extended Kalman Filter output, we used the Mean Absolute Error (MAE) metric,

which is the mean of the magnitude of the segments defined by the ground truth position and the estimated position.

The following tables report in detail the results of all the executed experiments. In particular, every table reports the MAE, referred to as ε, obtained by estimating the position of the mobile robot through an Extended Kalman Filter (EKF), the proposed feedforward neural network (FFNN) and the proposed LSTM neural network. Each MAE value is matched with its standard deviation σ. Finally, the percentage improvement brought by the FFNN and LSTM is reported.

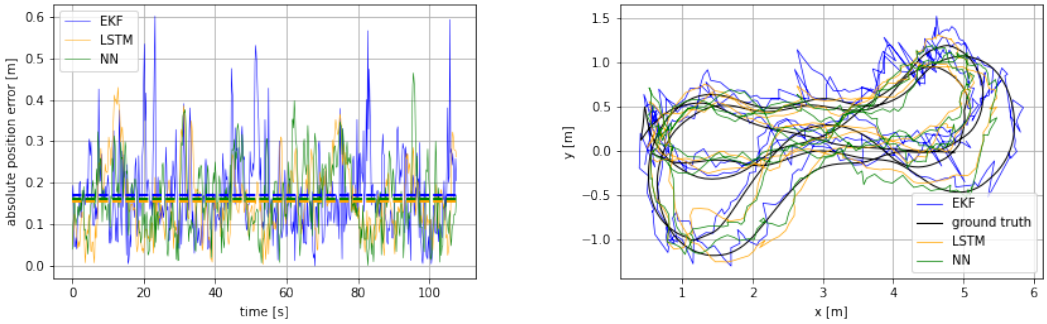
In all the reported graphs the black traces are referred to the ground truth data, the blue ones to the EKF estimation, the orange ones to the LSTM neural network estimation and the green ones to the FFNN estimation, as a comparison. The dashed lines, when present, represent the mean values of the considered error.

Experiment 1: Line of Sight

In this scenario, as said above, there are no obstacles between the four anchors displaced in the room and the tag fixed to the mobile robot. Considering the average value of the error obtained applying the three different solutions, in table **??**, it is immediately evident how both the NN-based solutions bring an improvement with respect to the EKF-based solution. In particular, the improvement of the LSTM network is, on average, 20 %. Moreover, even though the FFNN produces a slight improvement, its standard deviation increases by around 30 %, with respect to the other solutions, which are characterized by almost the same value.
Looking at figure 6.12b and analyzing it qualitatively, it can be noticed how the trajectory estimated by the LSTM network is significantly smoother than the other ones and can follow better the path of the ground truth.

n.	ε_{EKF}	ε_{FFNN}	ε_{LSTM}	σ_{EKF}	σ_{FFNN}	σ_{LSTM}	$\Delta^{\%}_{FFNN}$	$\Delta^{\%}_{LSTM}$
0	0.171	0.162	0.155	0.097	0.084	0.083	5.3	9.3
1	0.178	0.168	0.150	0.088	0.088	0.091	5.6	15.3
2	0.246	0.225	0.186	0.150	0.158	0.141	8.8	24.5
3	0.156	0.138	0.108	0.077	0.174	0.082	11.8	30.8
mean	0.185	0.177	0.146	0.100	0.129	0.097	8.1	20.8

Table 6.16: Results obtained in the evaluation of the localization error in LOS trajectory experiments. The last row refers to the overall mean.



(a) Position error during time. **(b)** Estimated trajectory.

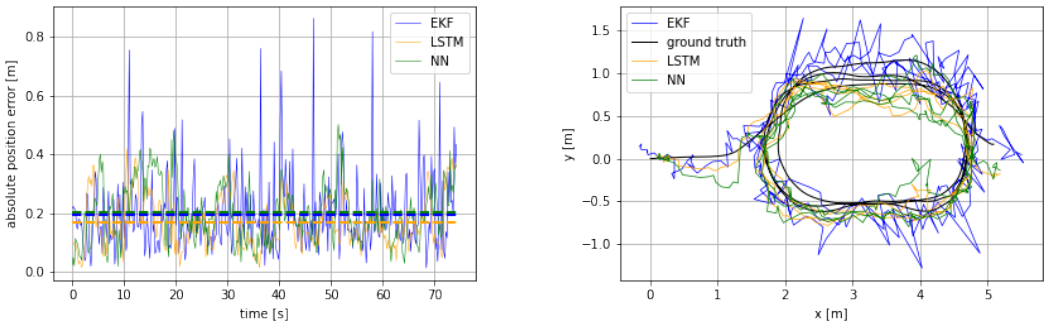
Figure 6.12: Graph representing the absolute position error during time and the estimated trajectory of experiment n. 0 in LOS condition.

Experiment 2: Non-Line of Sight

The NLOS situation is expected to be the worst-case scenario: in this case, indeed, at least one anchor is always covered by an obscale from the tag point of view. This can be clearly noticed also from the results: the error is much higher (around 50 - 70 %) that the LOS case. Also looking at the estimated position graph, in figure 6.13a, the EKF has significative overshoots and oscillations. In this case, both Neural Networks models struggle to achieve significative improvements. In particular, the FFNN model achieve almost the same results as the EKF, but the value of the standard deviation is definitely worse, meaning a more sparse distributon of the error. Nonetheless, the improvement achieved by the LSTM architecture is about 10 % with a similar value of standard deviation.

n.	ε_{EKF}	ε_{FFNN}	ε_{LSTM}	σ_{EKF}	σ_{FFNN}	σ_{LSTM}	$\Delta^{\%}_{FFNN}$	$\Delta^{\%}_{LSTM}$
4	0.230	0.214	0.191	0.125	0.209	0.156	6.9	16.6
5	0.193	0.202	0.168	0.124	0.100	0.086	-4.4	12.9
6	0.211	0.211	0.202	0.119	0.145	0.130	0.0	4.3
mean	0.212	0.209	0.189	0.122	0.154	0.126	1.1	10.5

Table 6.17: Results obtained in the evaluation of the localization error in NLOS trajectory experiments. The last row refers to the overall mean.



(a) Position error during time. **(b)** Estimated trajectory.

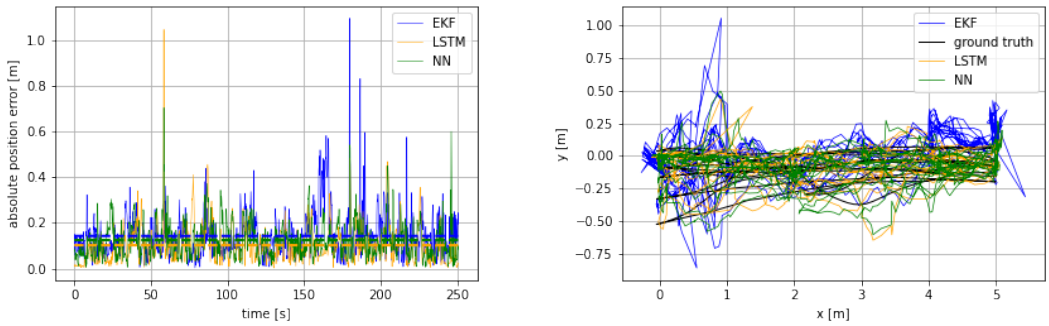
Figure 6.13: Graph representing the absolute position error during time and the estimated trajectory of experiment n. 5 in NLOS conditions. The trajectory estimated by the LSTM model is clearly less prone to larger errors.

Experiment 3: Realistic situation

This scenario offers a wide diversity of situations, in fact, it was recorded in dynamic situations with a changing environment, i. e. with moving people. Before looking at the achieved results, it must be said that, despite the more complex situation, it is the most similar to the training set of the neural networks and, therefore, the best improvement is expected. Analyzing the results, it can be noticed how both the NN models, generate a marked improvement with respect to the EKF outcome. In fact, the FFNN model brings an improvement of about 15 % while the LSTM model reaches a value of about 25 %. Also, the values of the standard deviations of both the NN models are neatly lower than the EKF one, meaning a tighter distribution of the error.

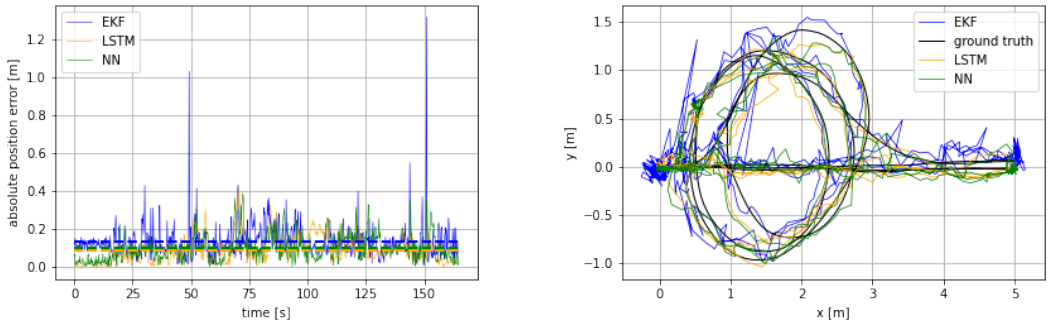
n.	ε_{EKF}	ε_{FFNN}	ε_{LSTM}	σ_{EKF}	σ_{FFNN}	σ_{LSTM}	$\Delta^{\%}_{FFNN}$	$\Delta^{\%}_{LSTM}$
7	0.142	0.124	0.102	0.097	0.078	0.079	12.4	28.4
8	0.134	0.102	0.090	0.085	0.069	0.070	23.5	32.6
9	0.127	0.117	0.107	0.081	0.071	0.071	7.8	15.7
10	0.205	0.138	0.127	0.107	0.096	0.107	32.5	37.9
mean	0.145	0.120	0.105	0.169	0.147	0.151	16.4	26.5

Table 6.18: Results obtained in the evaluation of the localization error in mixed LOS and NLOS trajectory experiments. The last row refers to the overall mean.



(a) Position error during time. **(b)** Estimated trajectory.

Figure 6.14: Graph representing the absolute position error during time and the estimated trajectory of experiment n. 7



(a) Position error during time. **(b)** Estimated trajectory.

Figure 6.15: Graph representing the absolute position error during time and the estimated trajectory of experiment n. 8

Chapter 7

Conclusions and future works

7.1 Conclusions

Considering the overall result in all the experiments, it is possible to state that neural networks, and in particular LSTM architectures, are able to estimate the encoder-inertial odometry and the UWB localization with a smaller error than the most common methods based on an Extended Kalman Filter. Concerning wheel odometry, in all the different experiments that were executed, an average improvement of around 60 % on the absolute positioning error and around 65 % of the absolute orientation error is achieved. In this case, it is not worthwhile to consider the absolute value of the error, because the difference between trajectories leads to highly different results. In all the experiments, the strongest improvement is observed in the short-term orientation prediction: this is the key to keeping all the other errors bounded. Since during the description of the problem the orientation uncertainty was identified as the first cause of positioning error and of the degradation of the whole trajectory estimation, its accuracy becomes fundamental to keep all the other errors bounded, especially the long-term ones. The final result is an improvement of around 30 % and 70 %, respectively on the cumulative errors of the position and orientation.

Regarding the UWB localization, the outcomes of the different experiments are a bit different among themselves. In every kind of scenario, the LSTM architecture outperformed the other types of approach. Particularly, even though the feedforward neural network achieves some slight improvement in the LOS and dynamic scenarios, it emerges its lacks of ability to tackle all the different situations, with the consequence that not always the position error improves, but its standard deviation

increases as well. The LSTM neural network, on the contrary, always achieve to provide better performance than the EKF. The main flaw of a neural network-based approach for UWB localization is the total dependence on the environment, which means that for a new room or a different anchor constellation, new training would be needed.

In both cases, from the grid search, it was demonstrated how a sequence of inputs works better than a single value, which implies that this kind of solution is more robust to outliers or, in UWB localization, biases due to NLOS situations. On the other hand, this work proved how two simple models (as the two LSTM networks with a single 96 neurons layer), are able to mitigate the error and offer an overall better performance than the EKF. For sure, this thesis can be intended as a starting point for the study of this topic using recurrent neural networks and time sequences.

7.2 Future works

This work can be intended as the first approach to a broader in-depth analysis or more concrete applications of the studied topic. Also other kinds of neural networks such as convolutional neural networks or auto-encoders can be investigated for this purpose. Regarding the encoder odometry, some future work can be devoted to:

- study and develop an online learning-based solution, to adapt in real-time to possible scenario changes, such as different floor materials, uneven terrains and surfaces or changes in the mobile robot's structure and payload.
- investigate the integration of different sensors' data, obtaining accurate wheel odometry during exteroceptive sensors outage or non-ideal conditions, such as in dark or scarcely illuminated environments for cameras, to provide a more robust solution for localization.

On the other hand, since the part regarding the UWB has started as side work, it would be interesting to continue to:

- study more in-depth, with different constellations of anchors and in a more wide set of Non-Line of Sight situations, the dynamic UWB localization using neural networks.
- Explore a sensor fusion approach for UWB localization and other kinds of sensors, to detect and mitigate the Non-Line of Sight conditions, when detected.

Bibliography

[1] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza. *Introduction to Autonomous Mobile Robots - 2nd ed.* The MIT press, 2011. ISBN: 0262015358 (cit. on pp. 4–7, 15, 48–50, 52).

[2] P. K. Panigrahi and S. K. Bisoy. «Localization strategies for autonomous mobile robots: A review». In: *2009 International Conference on Machine Learning and Applications* 34 (8 2022), pp. 6019–6039. DOI: https://doi.org/10.1016/j.jksuci.2021.02.015 (cit. on pp. 5, 14, 16).

[3] J. Borenstein and L. Feng. «UMBmark: a benchmark test for measuring odometry errors in mobile robots». In: *Proceedings of the SPIE*. Vol. 2591. 1995, pp. 113–124. DOI: 10.1117/12.228968 (cit. on pp. 7, 19).

[4] T. Zwick, W. Wiesbeck, J. Timmermann, and G. Adamiuk. *Ultra-wideband RF System Engineering*. Vol. 53. Cambridge University Press, Jan. 2013. DOI: 10.1017/CBO9781139058957 (cit. on pp. 8–10).

[5] S. Angarano. «Deep Learning Methodologies for UWB Ranging Error Compensation». MSc thesis. Torino, Italy: Politecnico di Torino, 2020 (cit. on pp. 10, 11).

[6] Sherif A. S. Mohamed, Mohammad-Hashem Haghbayan, Tomi Westerlund, Jukka Heikkonen, Hannu Tenhunen, and Juha Plosila. «A Survey on Odometry for Autonomous Navigation Systems». In: *IEEE Access* 7 (2019), pp. 97466–97486. DOI: 10.1109/ACCESS.2019.2929133 (cit. on p. 17).

[7] Haoming Xu and John James Collins. «Estimating the Odometry Error of a Mobile Robot by Neural Networks». In: *2009 International Conference on Machine Learning and Applications*. 2009, pp. 378–385. DOI: 10.1109/ICMLA.2009.96 (cit. on p. 19).

[8] Uche Onyekpe, Vasile Palade, Stratis Kanarachos, and Stavros-Richard G. Christopoulos. «Learning Uncertainties in Wheel Odometry for Vehicular Localisation in GNSS Deprived Environments». In: *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 2020, pp. 741–746. DOI: 10.1109/ICMLA51294.2020.00121 (cit. on p. 19).

[9] Martin Brossard and Silvère Bonnabel. «Learning Wheel Odometry and IMU Errors for Localization». In: *2019 International Conference on Robotics and Automation (ICRA)*. 2019, pp. 291–297. DOI: 10.1109/ICRA.2019.8794237 (cit. on p. 20).

[10] N.Carlevaris-Bianco, A. K. Ushani, and R. M. Eustice. «University of Michigan North Campus long-term vision and lidar dataset». In: *International Journal of Robotics Research* 35.9 (2015), pp. 1023–1035. DOI: https://doi.org/10.7302/7rnm-6a03 (cit. on p. 20).

[11] K. O. Canbek, H. Yalcin, and E. A. Baran. «Drift compensation of a holonomic mobile robot using recurrent neural networks». In: *Intel Serv Robotics*. Vol. 5. 2022, pp. 399–409. DOI: 10.1007/s11370-022-00430-w (cit. on p. 20).

[12] Simone Angarano, Vittorio Mazzia, Francesco Salvetti, Giovanni Fantin, and Marcello Chiaberge. «Robust ultra-wideband range error mitigation with deep learning at the edge». In: *Engineering Applications of Artificial Intelligence* 102 (2021), p. 104278. ISSN: 0952-1976. DOI: https://doi.org/10.1016/j.engappai.2021.104278 (cit. on p. 20).

[13] Alwin Poulose and Dong Seog Han. «UWB Indoor Localization Using Deep Learning LSTM Networks». In: *Applied Sciences* 10.18 (2020). ISSN: 2076-3417. DOI: 10.3390/app10186290 (cit. on p. 21).

[14] Alwin Poulose and Dong Seog Han. «Feature-Based Deep LSTM Network for Indoor Localization Using UWB Measurements». In: *2021 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*. 2021, pp. 298–301. DOI: 10.1109/ICAIIC51459.2021.9415277 (cit. on p. 21).

[15] Wenda Zhao, Jacopo Panerati, and Angela P. Schoellig. «Learning-Based Bias Correction for Time Difference of Arrival Ultra-Wideband Localization of Resource-Constrained Mobile Robots». In: *IEEE Robotics and Automation Letters* 6.2 (2021), pp. 3639–3646. DOI: 10.1109/LRA.2021.3064199 (cit. on p. 21).

[16] Doan Tan Anh Nguyen, Jingon Joung, and Xin Kang. «Deep Gated Recurrent Unit-Based 3D Localization for UWB Systems». In: *IEEE Access* 9 (2021), pp. 68798–68813. DOI: 10.1109/ACCESS.2021.3077906 (cit. on p. 22).

[17] E. Kavlakoglu. *AI vs. Machine Learning vs. Deep Learning vs. Neural Networks: What's the Difference?* May 2020. URL: https://www.ibm.com/cloud/blog/ai-vs-machine-learning-vs-deep-learning-vs-neural-networks (cit. on pp. 23, 24).

[18] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. The MIT press, 2017. ISBN: 0262035618, 9780262035613 (cit. on pp. 25, 33).

[19] B. Blaus. URL: https://en.wikipedia.org/wiki/Neuron#/media/File:Blausen_0657_MultipolarNeuron.png (cit. on p. 26).

[20] A. Géron. *Hands-on machine learning with Scikit-Learn, Keras and TensorFlow: concepts, tools, and techniques to build intelligent systems (2nd ed.)* O'Reilly, 2019. ISBN: 9781098125974 (cit. on pp. 27, 43, 45–47).

[21] A. Chandra. *McCulloch-Pitts Neuron — Mankind's First Mathematical Model Of A Biological Neuron*. 2018. URL: https://towardsdatascience.com/mcculloch-pitts-model-5fdf65ac5dd1 (cit. on p. 27).

[22] G. Calafiore. «(Linear) Suppert Vector Machines». Lecture slides. Politecnico di Torino. 2021 (cit. on p. 28).

[23] M.A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015 (cit. on pp. 31, 32, 36, 38, 40).

[24] G. Calafiore. «Feed-Forward Neural Networks». Lecture slides. Politecnico di Torino. 2021 (cit. on pp. 31, 35).

[25] D. P. Kingma and J. Ba. *Adam: A Method for Stochastic Optimization*. 2014. DOI: 10.48550/ARXIV.1412.6980 (cit. on pp. 42, 70).

[26] A. Graves. *Supervised Sequence Labelling with Recurrent Neural Networks*. Springer Berlin, Heidelberg, 2012. ISBN: 978-3-642-24796-5 (cit. on pp. 43–45).

[27] X. Glorot and Y. Bengio. «Understanding the difficulty of training deep feedforward neural networks». In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* 9 (2010), pp. 249–256 (cit. on p. 45).

[28] A. Rizzo. «Mobile & Service Robotics Introduction». Lecture slides. Politecnico di Torino. 2021 (cit. on pp. 48, 51).

[29] B. Siciliano, L. Sciavicco, L. Villani, and G. Orlo. *Robotica. Modellistica, pianificazione e controllo*. Mc Graw Hill Education, 2008. ISBN: 883866322X (cit. on p. 51).

[30] *Intel RealSense Tracking Camera*. Intel RealSense. 2019. URL: https://www.intelrealsense.com/tracking-camera-t265/ (cit. on pp. 53, 54).

[31] *Leica Absolute Tracker AT403 di Hexagon*. URL: https://www.diati.polito.it/il_dipartimento/laboratori/geomatics_lab/fotogrammetria_e_gis/leica_absolute_tracker_at403_di_hexagon (cit. on p. 55).

[32] *Leica Laser Tracker At403*. 2021. URL: https://www.metrologyshare.com/laser-tracker/leica-laser-tracker-at-403/ (cit. on p. 55).

[33] *TREK1000 User Manual*. 2016. URL: https://datasheet.octopart.com/TREK1000-Decawave-datasheet-98959800.pdf (cit. on p. 57).

[34] *Jackal datasheet*. Clearpath robotics. 2014. URL: https://clearpathrobotics.com/jackal-small-unmanned-ground-vehicle/ (cit. on pp. 58, 59).

[35] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall. «Robot Operating System 2: Design, architecture, and uses in the wild». In: *Science Robotics* 7.66 (2022), eabm6074. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074 (cit. on p. 60).