Politecnico di Torino

Master of Science Degree in Mechatronic Engineering

Master Thesis

Indoor SLAM and Room Classification with Deep Learning at the edge

Supervisor:
prof: Marcello Chiaberge

Candidate:
Andrea Eirale
S267878

Academic year 2019-2020

Abstract

In recent years, the development of technology has led to the emergence of increasingly complex and accurate localization and mapping algorithms. In the field of robotics, this has allowed the progressive integration alongside other programs with the most varied functions, from home care to space exploration, designed to provide a service, to support and improve people's living conditions. With this goal in mind, the PIC4SeR (PoliTo interdepartmental centre for service robotics) has developed the idea of integrating a SLAM algorithm, for simultaneous self localization and mapping, with a convolutional neural network for room recognition.

The main goal of this thesis project is the development of an algorithm able to lead an unmanned ground vehicle in an unknown, closed domestic environment, mapping it and classifying each room encountered in the process. Low cost sensors and free, open-source software are used to achieve the final result.

For localization and mapping, several techniques are considered, from the classic extended Kalman filter to the more advanced graph-based SLAM. The most adapted ones are further developed, to retrieve a first, raw representation of the environment.
The map is then processed with computer vision software in order to obtain a cleaner and clearer plot of the surroundings, and to setting it up for the recognition algorithm.

Finally, a convolutional neural network model is used, alongside to a series of frame images taken by the robot from the environment, to classify each room and provide predictions on the map.

The final algorithm is relatively efficient and lightweight, and opens up to a series of future implementations in the field of service robotics, in domestic environments and in the assistance to elderly and disabled users.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Objective of the thesis

In recent years, the development of technology has led to the emergence of increasingly complex and accurate localization and mapping algorithms. In the field of robotics, this has allowed the progressive integration alongside other programs with the most varied functions, from home care to space exploration, designed to provide a service to support and improve people's living conditions. With this goal in mind, the PIC4SeR (PoliTo interdepartmental centre for service robotics) has developed the idea of integrating a SLAM algorithm, for simultaneous self localization and mapping, with a convolutional neural network for room recognition.

The main goal of this project of thesis is to develop an algorithm able to lead an unmanned ground vehicle in an unknown, closed domestic environment, mapping it and classifying each room encountered in the process. Low cost sensors and free, open-source software are used to achieve the final result. For localization and mapping part, several techniques are introduced in this paper, starting from the historical ones to the more advanced algorithms used nowadays, analyzing merits and complications of each one. The most common Machine Learning classification algorithms are briefly exposed, focusing more deeply on Deep Learning techniques and explaining what is a Neural Network, how it works, and how is used for image classification. Later, the main implementation used to achieve the goal of the project are presented, both for the localization and mapping and for classification, pointing out difficulties encountered and solutions adopted. Finally, a short record about future improvements and integration is reported, stating how this project could be used for further implementations.

1.2 Organization of the thesis

The thesis is composed of seven chapters, organized as follows:

- The first chapter is an introduction to the project, presenting the main motivations and goals, and briefly explaining the topics covered in the following sections;
- The second chapter is a theoretical introduction to Machine Learning and its most common classification techniques. Deep Learning Neural Networks are later explained more in details, how they work and why are so important for image classification, and then for the room recognition problem;
- The third chapter briefly presents what is a robot, how it is designed and used, depending on the tasks it has to perform and the characteristic of the environment of work, with a major focus on the mobile, wheeled robot, which is the type employed in this project. Then it passes to expose the main sensors commonly used on such robots and how they works. Later, the *Robotic Operative System* (ROS) is introduced, as one of the main and more diffused means to control and organize a robotic system. Finally, the hardware components used on the robot in this thesis project are presented, justifying why they where chosen instead of others;
- The fourth chapter is a theoretical introduction to SLAM, involving Localization and Mapping in addition to the major filters used to solve such problems. Then the most known SLAM technique are presented, from the historical Extended Kalman Filter to the more advanced and at the State-of-Art Graph-Based simultaneous localization and mapping;
- The fifth chapter presents the implementation of both, the classification and the SLAM algorithms used in the project, how they are created and improved, with the major difficulties encountered, and how they are integrated in the final localization, mapping and classification program;
- The sixth chapter sums up the entire work done in this thesis project, and show the results of every section;
- The seventh chapter conclude the paper, exposing how outcomes obtained by this work can be improved and presenting a series of practical applications for which this project can be useful.

Chapter 2

Machine Learning

Classification is a statistical problem much older than the invention of Machine Learning, but which, in recent years, has received a great benefit from technological advancement and, in particular, from the development of so-called *Deep Learning*. In order to better understand what a classification problem really is, and how and why deep learning is so effective in solving it, it is necessary to introduce the basic principles of machine learning.

2.1 Machine Learning basics

Machine Learning is essentially a form of applied statistic which consists in the use of computers to estimate complex functions. The main goal is to allow an algorithm to *learn* from a certain type of data, and use such information to predict new instances. To better comprehend what is meant when it is said that a computer algorithm can learn, Tom M. Mitchell provided the following definition: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E" [1]. In this quote, it is possible to extrapolate three variables, which are here better explained.

The Task, T How said before, machine learning is used to solve complex functions, which are virtually impossible to be computed with programs written and designed by human beings. In these problems, learning itself is not the task of the algorithm, learning is indeed the means by which this task is accomplished. Generally, in machine learning these tasks are referred to in terms of how the algorithm has to process a certain number of instances. This directly affect the format of the input data, how and what the algorithm has to learn from it, and how has to use such information to give a result. Classification is one of several tasks which can be solved with machine learning, and will be examined more deeply later on.

The Performance Measure, P To evaluate how well a machine learning algorithm is performing, it is necessary to define a quantitative measure. Such measure is specific to the task T to be carried out. For example, in a classification problem the variable to be evaluated is generally the *accuracy*, based on the correctness of predictions preformed. On the other side, it is possible to obtain the same information evaluating the *error rate*, that is based on incorrect output produced.

Usually, these performance measures are obtained by testing the model on a set of data it has never seen. This set is generally called the *test set*, in opposition to the set used for training, the *training set*. Therefore the test set is used to evaluate how well the model will perform in the real world. However, the choice of the test data is not always so easy, and often it depends on the kind of application implemented.

The Experience, E Most of the learning algorithms, classification ones included, are designed to experience from an entire *dataset*, a collection of many instances, but a further distinction can be done:

- An **Unsupervised learning algorithm** experiences a dataset containing many features and learns, by itself, the structure's properties of such dataset. From a practical point of view, the algorithm has to observe several instances of a random vector $\boldsymbol{x}$, and learn to predict $p(\boldsymbol{x})$, the probability distribution of $\boldsymbol{x}$;
- A **Supervised learning algorithm** experiences a dataset containing many feature, but each instance is also associated to a *label*. All the labels together form the structure of the dataset, which is so provided. From a practical point of view, the algorithm has to observe several instances of a random vector $\boldsymbol{x}$ and the associated vector of labels $\boldsymbol{y}$, then learn to predict $\boldsymbol{y}$ from $\boldsymbol{x}$, generally by estimating $p(\boldsymbol{y}|\boldsymbol{x})$, the probability distribution of $\boldsymbol{x}$ in relation to $\boldsymbol{y}$.
- There are then machine learning algorithms which do not simply experience a dataset. An example is the **Reinforcement learning algorithm**, that interact with an environment with a feedback loop between the learning system and its experiences.

2.1.1 Overfitting and Underfitting

How said before, to evaluate the performance of a machine learning model, often the errors are taken into consideration. More precisely, it is very important to consider both, the errors on training and validation set. How it is well evident, a model is performing well when these two errors are both sufficiently low. Less obvious is the fact that the gap between these two values has to be small. In fact, as stated before, a machine learning model is preforming well when it is able to correctly predict new instances it has never seen before. This behavior is called

generalization, because the model manages to generalize what it has learned, and does not stagnate on the data used for training. This ability is directly connected with the *capacity* of the model, which is the ability to fit a wide variety of functions.

When the capacity of a model is too high, there is an high probability of incurring in the phenomenon of *Overfitting*. In particular, overfitting occur when the model learn to predict the data contained in the training set extremely well (it is said that the model *specializes* on the training set) and, as a consequence, it is unable to generalize on other instances, namely it is inefficient in estimate any other instance that does not belong to the training set. Practically, this event is recognizable when the training error is very low, and the error on the test set is much higher.

On the opposite side, when the capacity of the model is too low, the algorithm is not able to fit the training set, and arise a problem of *Underfitting*. In this case, there is still room for improvement on the training set, which present an error unacceptably high, independently from the test's error.

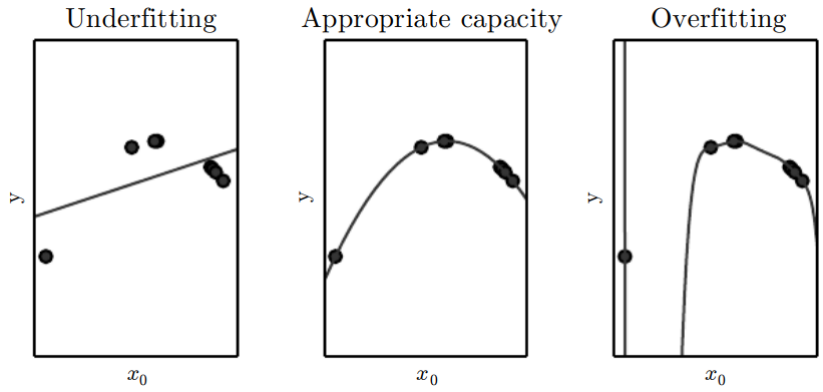


Figure 2.1: Visualization of Overfitting and Underfitting [2].

Machine learning algorithm will generally performs better when their capacity is appropriate to the task they need to accomplish. To solve the overfitting and underfitting problems, a series of techniques exist, some of which will be discussed in this paper later on.

2.1.2 Validation

In the implementation of a machine learning algorithm, in addition to the training and test set, there is another fundamental dataset called *Validation set*. While the concept of test set and validation set can be easily mistaken, they are two different things: both these sets contain data the model has never seen before, hence

different from the training set, but if the former is the set of data used to evaluate the performance of a fully-specified model, the latter is the dataset used to tune the parameters of such model during its implementation. Typically, the validation set is obtaining dividing an initial dataset in two, with a 75/80% used as training set and the remaining 20/25% used for validation. Machine learning models have several settings used to control the behavior of the learning algorithm, and the validation set can be used to test and set these settings, called *hyperparameters*.

2.1.3 Linear Regression

In machine learning, *Linear Regression* is one of the most simple algorithm, and is at the base of any machine learning algorithm. How the name implies, it is specialized in solving regression problems, a task in which the output is in the form of continuous numerical values. For this reason, linear regression is not particularly suitable for classification problems, but it can be useful to better understand the classification techniques explained in the next section.

From a practical point of view, the task of linear regression is to take as input a vector $\boldsymbol{x}$, and predict the scalar value $\hat{y}$, associated with the ground truth scalar value y, which is a linear function of the input, defined as:

$$\hat{y} = f(\boldsymbol{x}) = \boldsymbol{\theta} \cdot \boldsymbol{x} \tag{2.1}$$

where $\boldsymbol{\theta}$ is a vector of parameters, sometimes also called *weight* vector and indicated with $\boldsymbol{w}$. Each parameter θ_i affects the corresponding feature x_i in a different manner.

In other words, the goal of linear regression is to estimate the probability distribution of y in function of $\boldsymbol{x}$ and $\boldsymbol{\theta}$, $p(y|\boldsymbol{x}, \boldsymbol{\theta})$, such that:

$$p(y|\boldsymbol{x}, \boldsymbol{\theta}) = f(\boldsymbol{x}) = \boldsymbol{\theta} \cdot \boldsymbol{x} \tag{2.2}$$

The performance of such model can be evaluated by computing the *mean squared error* of a test set, defined as:

$$MSE = \frac{1}{m} \sum_{i=1}^{m} (\hat{\boldsymbol{y}} - \boldsymbol{y})_i^2 \tag{2.3}$$

where m is the number of instances of the test set, $\hat{y}$ is the predicted value and y is the ground truth value. Hence, the goal is to design a model able to learn to optimize the parameters θ during the training phase, in a way that reduces the MSE. To minimize such cost function, there is a closed-form solution that give the results directly, called *Normal Equation*:

$$\hat{\boldsymbol{\theta}} = (\boldsymbol{X}^T \boldsymbol{X})^{-1} \boldsymbol{X}^T \boldsymbol{y} \tag{2.4}$$

where $\hat{\boldsymbol{\theta}}$ is the value of $\boldsymbol{\theta}$ that minimize the cost function and $\boldsymbol{X}$ is the training set.

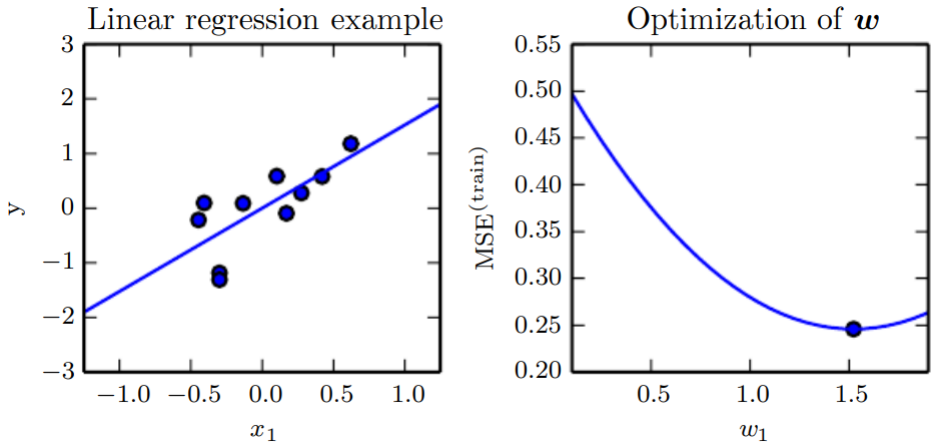


Figure 2.2: On the left, the linear regression optimized the weight vector, such that the function $y = w_i x$ passes as close as possible to instances points. On the right, the point indicate the value w_i, found by the normal function which minimize the mean squared error.

2.2 The Classification Task

Classification is the statistical problem of identifying the category membership of each new instance from a set of categories established a priori, starting from a training set of correctly identified observations. Classification can be divided into two types:

- **Binomial (or Binary)**: where instances belong only to two classes. Typically a one-vs-rest strategy is involved, in which case one of the two classes represent the normal state (a positive response, generally identified by the label 0) and the other the abnormal state (a negative response, identified by label 1). The two different type of error (false positive and false negative) are usually treated with different importance weight. Some algorithms, like Logistic Regression and Support Vector Machines, are specifically designed for binary classification and do not support multi-class problems unless they are turned by a variety of strategies;
- **Multi-Class**: where more than two classes are present. Different strategies are used in order to solve such kind of problems, and generally are categorized into transformation to binary, extension from binary and hierarchical classification. In the sector of machine learning, and specifically in that of neural network, multi-class tasks are solved exploiting N binary neurons in the final output layer, one for each class, instead of just one, and normalizing with a softmax function.

How said in the previous section, in order to evaluate the quality of a machine

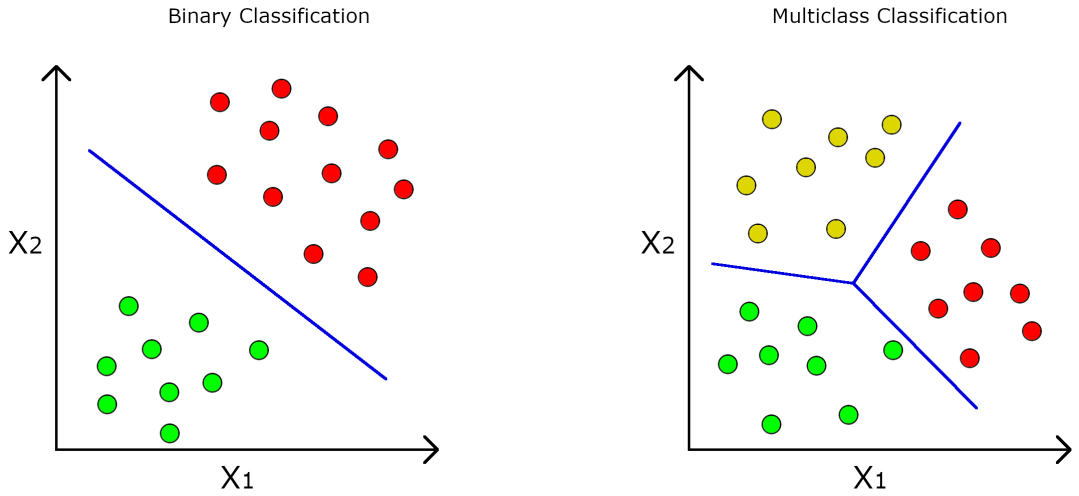


Figure 2.3: Graphical representation of the difference between Binary and Multiclass Classification.

learning model, some performance measures are necessary, and for a classification model these are made through two main parameters, *accuracy* and *loss*. These metrics are generally considered not only on the training set, but also on the *validation set*, and are defined as follows:

- Accuracy is expressed as a percentage, and is defined as ratio between the number of correctly predicted samples of the training (or validation) set and the total number of samples in the training (or validation) set. It is well evident that this value has to be as high as possible. Generally, values of accuracy equal or greater than 70% are acceptable results, but it depends on the case at hand a lot;

- A loss function is used to optimize a machine learning algorithm. Contrary to accuracy, loss value is not expressed as a percentage, and is defined as the sum of errors made for each sample in training (or validation) set. Practically, loss value implies how poorly or well a model behaves after each iteration, and has to be as low as possible.

In a classification problems, overfitting occur when the accuracy on the training set continues to increase reaching high values, often near to 100%, while the accuracy on validation set stagnates on low values or even decreases through each samples. At the same time, it can happen that loss on the training set reaches very low value, near to zero, while on validation set it stagnates on high values or even increases through each samples. On the other side, underfitting occur when the accuracy on the training set is too low.

2.3 Machine learning classification

In machine learning, classification is considered as an example of supervised learning, and a series of classifier algorithms have been developed during the years. In this section, the most popular ones are briefly exposed.

2.3.1 Logistic Regression

While linear regression is not suitable for classification, there are regression algorithms which can be used for classification as well. One of these is the *Logistic Regression*, one of the most used algorithms for binary classification, but that can also be extended for multi-class problems. The task is pretty similar to that of linear regression, to compute a weighted sum of the input features, but contrary to before, the output has to be the logistic of the result:

$$p(y|\boldsymbol{x}, \boldsymbol{\theta}) = f(\boldsymbol{x}) = \sigma(\boldsymbol{x}^T \boldsymbol{\theta}) \tag{2.5}$$

Unlike Linear Regression, no closed form solutions can solve the problem, so the logistic σ is a *sigmoid function* which outputs a number between 0 and 1:

$$\sigma(t) = \frac{1}{1 + e^{-t}} \tag{2.6}$$

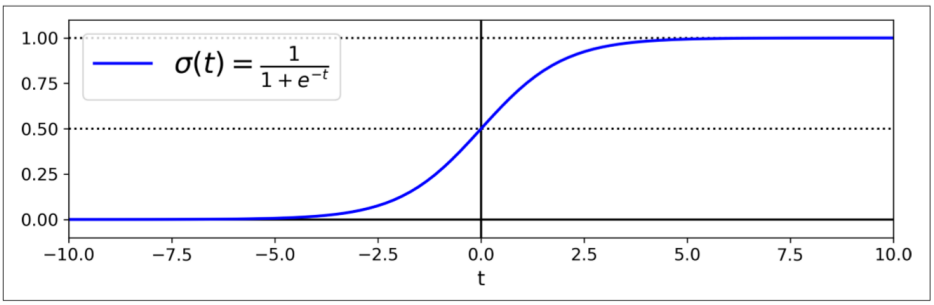


Figure 2.4: Logistic function [3].

Therefore, the logistic regression model estimates the probability p that an instance $\boldsymbol{x}$ belongs to the positive class or not, then makes its prediction $\hat{y}$:

$$\hat{y} = \begin{cases} 0 & \text{if} \quad p < 0.5, \\ 1 & \text{if} \quad p \geq 0.5 \end{cases} \tag{2.7}$$

The logistic regression can be generalized to be used for multi-class problems without having to combine multiple binary classifiers. This generalization takes the name of *Softmax Regression*. For each instance $\boldsymbol{x}$, it computes a score $s_k(\boldsymbol{x})$ for every class k, then estimates the probability of each class by applying the softmax

function to te scores. The computation of the score is very similar to the equation for linear regression prediction:

$$s_k(\boldsymbol{x}) = \boldsymbol{x}^T \boldsymbol{\theta}_k \tag{2.8}$$

It is worth noting that each class has its own parameter vector $\boldsymbol{\theta}_k$. Once all the scores are obtained, the softmax function is applied:

$$p_k = \sigma(\boldsymbol{s}(\boldsymbol{x}))_k = \frac{\exp(s_k(\boldsymbol{x}))}{\sum_{j=1}^{K} \exp(s_j(\boldsymbol{x}))} \tag{2.9}$$

The softmax function estimates the probability p_k, which is equal to $\sigma(\boldsymbol{s}(\boldsymbol{x}))_k$, the estimated probability that the instance $\boldsymbol{x}$ belongs to the class k given the vector $\boldsymbol{s}(\boldsymbol{x})$, containing the scores of each class for that instance; and does this by computing the ratio between the exponential of the score of instance $\boldsymbol{x}$ for class k, and the summation of exponentials of the score of instance $\boldsymbol{x}$ for all the classes K.

Just as the logistic regression, the softmax regression predicts the class with the highest estimated probability, which is the class with the highest score:

$$\hat{y} = \underset{k}{\operatorname{argmax}}(\sigma(\boldsymbol{s}(\boldsymbol{x}))_k) = \underset{k}{\operatorname{argmax}}(s_k(\boldsymbol{x})) = \underset{k}{\operatorname{argmax}}((\boldsymbol{\theta}_k)^T \boldsymbol{x}) \tag{2.10}$$

2.3.2 Support Vector Machine (SVM)

A SVM construct one or more hyperplanes in a high-dimensional space such that these have the largest distance to the nearest training data point of any class. A hyperplane is then a set of points $\boldsymbol{x}$ which satisfy the function:

$$\boldsymbol{w} \cdot \boldsymbol{x} + b = 0 \tag{2.11}$$

Where $\boldsymbol{w}$ is the normal vector to the hyperplane and $\frac{b}{||\boldsymbol{w}||}$ is the offset of the hyperplane from the origin. Anything below

$$\boldsymbol{w} \cdot \boldsymbol{x} + b < 0$$

belongs to one class, and anything above

$$\boldsymbol{w} \cdot \boldsymbol{x} + b \geq 0$$

belongs to the other class.

Kernel Support Vector Machine is similar to SVM but, contrary to this, it is used for nonlinear classification problems. This is possible by projecting the nonlinear data to an higher dimensional space, where it is linearly separable by a hyperplane, which obtain a nonlinear shape returning to the original dimensional space.

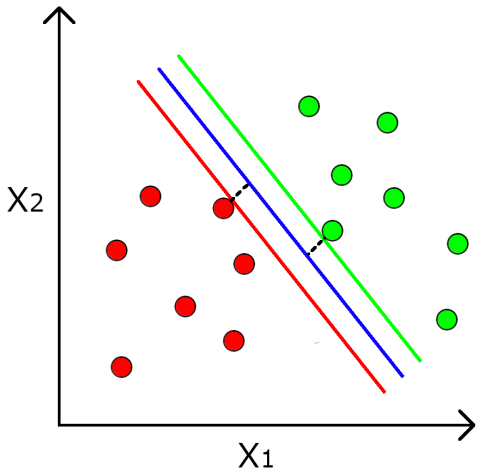


Figure 2.5: Support Vector Machine representation.

2.3.3 K-nearest Neighbors (KNN)

In KNN classification, a number of neighbors K is chosen a priori and then, for each instance $\boldsymbol{x}$, the algorithm identifies the k closest neighbors from that instance, and provide a prediction based on the majority vote of these neighbors. The algorithm searches for nearest neighbors of an instance $\hat{x}$ employing the *Minkowski* metric:

$$||\hat{x} - x_j||^p = (\sum_{i=1}^{q} |(\hat{x_i}) - (x_i)_j|^p)^{1/p} \tag{2.12}$$

Which, for $p = 2$, correspond to the Euclidean distance. In case of binary classification, the KNN algorithm is defined as:

$$f_{\text{KNN}}(\hat{x}) = \begin{cases} 1 & \text{if} \quad \sum_{i \in N_k(\hat{x})} y_i \geq 0, \\ -1 & \text{if} \quad \sum_{i \in N_k(\hat{x})} y_i < 0 \end{cases} \tag{2.13}$$

For a multiclass classification problem, this equation become:

$$f_{\text{KNN}}(\hat{x}) = \underset{y \in \boldsymbol{y}}{\text{argmax}} \sum_{i \in N_k(\hat{x})} I(y_i = y) \tag{2.14}$$

With the function $I()$ which return 1 if its argument is true and 0 otherwise. This algorithm is an example of lazy learning, since the function is only approximated locally, and normalizing the training data can improve the accuracy [4].

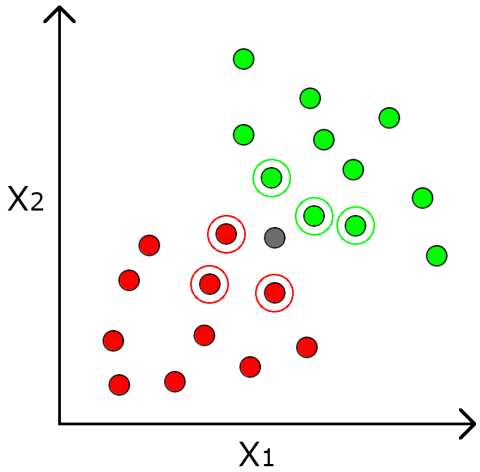


Figure 2.6: K-Nearest Neighbors representation.

2.3.4 Naive Bayes

Naive Bayes classifiers works on the principle of conditional probability as given by Bayes theorem, which can be expressed as follows:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{2.15}$$

where A and B are events and P(B)≠0:

- $P(A)$ is the probability of occurrence of event A;
- $P(B)$ is the probability of occurrence of event B.
- $P(A|B)$ is the probability of occurrence of event A when B is true;
- $P(B|A)$ similarly is the probability of occurrence of event B when A is true;

In a binomial classification problem, in which each new instance has to be assigned to class A or B, in order to apply a Naive Bayes classifiers, for each new instance a number of parameters are needed:

- $P(A)$: probability of occurrence of event A, which is the ratio between the number of observations in the class A and the total number of observations;
- $P(B)$: probability of occurrence of event B, which is the ratio between the number of observations in the class B and the total number of observations;
- $P(X)$: marginal probability, which is the ratio between the number of observations within a certain radius from a new instance and the total number of observations;

- $P(X|A)$: conditional probability that observations of class A exhibit features of X, which is the ratio between observations of class A within a certain radius from a new instance and the total number of observations of class A;
- $P(X|B)$: conditional probability that observations of class B exhibit features of X, which is the ratio between observations of class B within a certain radius from a new instance and the total number of observations of class B;

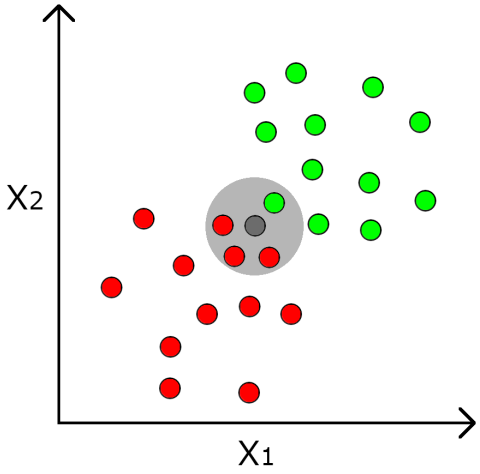


Figure 2.7: Naive Bayes representation.

Then Bayes filter is applied as follows:

$$P(A|X) = \frac{P(X|A)P(A)}{P(X)}$$

and

$$P(B|X) = \frac{P(X|B)P(B)}{P(X)}$$

Finally the two results $P(A|X)$ and $P(X|B)$ are compared and on this basis the new instance is assigned to the more probable class.

2.3.5 Decision Tree and Random Forest Classification

Decision Tree Classifiers have been very used in the past due to their qualities, among the others for being very easy and intuitive to implement (require very little data preparation) but at the same time very powerful algorithms. They are also the fundamentals for Random Forest Classifier. A Decision Tree Classifier, like the name says, lean on a Decision Tree, which is traversed for each instance to find the corresponding leaf node, and then returns the probability of belonging to each class.

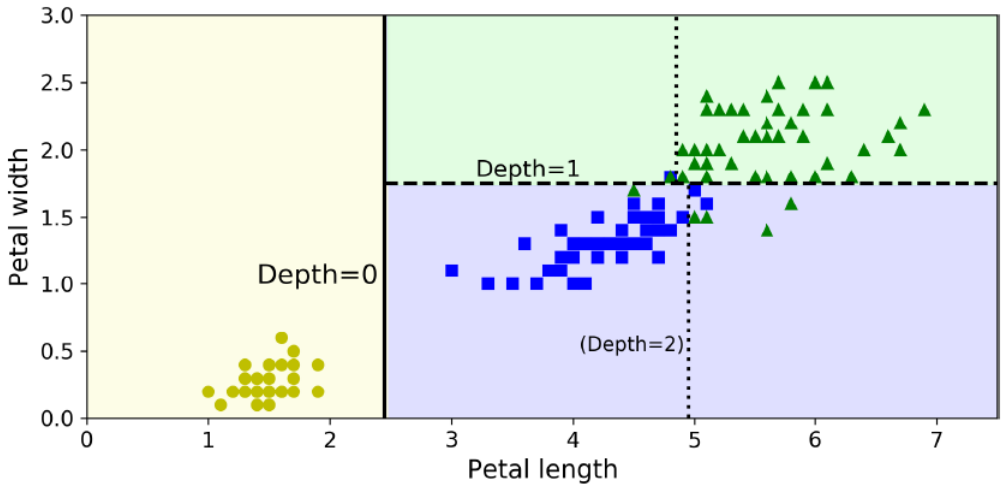


Figure 2.8: Graphical representation of Decision Tree Classification boundaries on an example problem involving the recognition of flowers on the base of width and length of their petals [3].

Random Forest is one of the most powerful machine learning algorithms nowadays that operate by aggregating the predictions of a group of predictors, in particular of a multitude of decision trees. Practically, Random Forest classifier pick K random data points from the training set and construct a decision tree on these points. This operation is repeated N time, where N is the number of tree desired. When a new instance is received, every tree constructed in such a way make a prediction, and the new data point is assigned to the class that wins the majority vote.

2.4 Deep learning classification

2.4.1 The artificial neuron

Deep learning is a branch of machine learning based on artificial neural networks, of which neurons are the fundamental elements. Neurons take a certain number of weighted input element, sum theme up and then apply an *activation function* in order to obtain an output. There are lots of different activation function, and depending on the problem to be solved some do the job better than others. The most used are the Threshold (step) function and the Rectifier function (also called *ReLU*), that will be presented later on in this paper, the sigmoid function, already defined in (2.6), and the Hyperbolic tangent function.

The most simple example of artificial neuron, and also one of the first theorized, is the *perceptron*, developed in the 1950s and 1960s by the scientist Frank Rosenblatt, inspired by earlier work by Warren McCulloch and Walter Pitts. A perceptron takes several binary inputs, each associated with a different weight, and produce a single binary output, that follows the threshold activation function below:

$$\text{output} = \begin{cases} 0 & \text{if} \quad \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if} \quad \sum_j w_j x_j > \text{threshold} \end{cases} \tag{2.16}$$

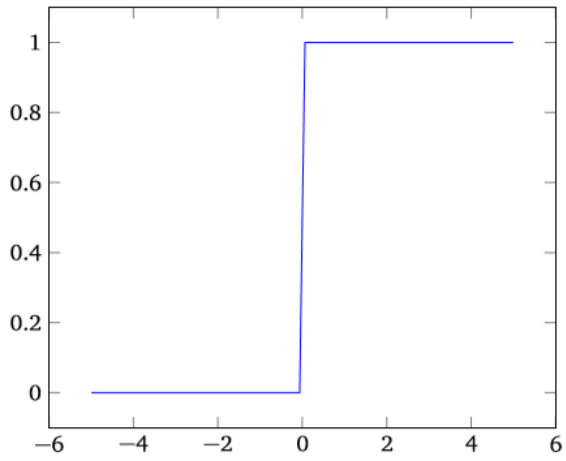


Figure 2.9: Step function [5].

The main limit of perceptrons is that a small change in the weights or in the threshold (generally called *bias*) could cause heavy and unpredictable changes in the output, since a characteristic property of perceptrons is that they can only assume value 0 or 1. Considering that small changes in weights is an operation

of primary importance in the process of learning, this throwback can create some very complicated scenarios.

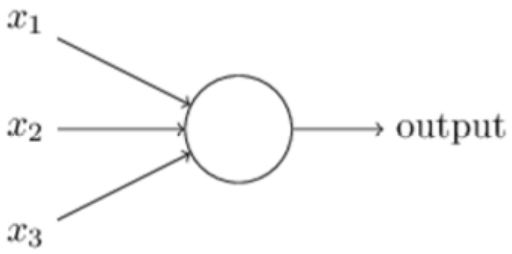


Figure 2.10: Perceptron representation [5].

This problem can be overcome by using *Sigmoid* neurons. These are very similar to perceptrons, but in addition to assume value 0 or 1, they can contain any other value between these two. Consequently the output of a Sigmoid neuron will not be binary, but $\sigma(wx + b)$, where σ is the sigmoid activation function (2.6).

2.4.2 Artificial Neural Networks

An artificial neural network is then formed by a series of neurons organized in layers, and each layer is fully connected to one another. Generally three types of layers are present:

- An **Input layer**, where is initially contained the data entering the network;
- An **Output layer**, the last layer of the network which provide the final output data;
- A certain number of **Hidden layers**, so called because are between the previous two layers. Networks with an high number of hidden layers are usually more complex and can achieve better decision making results.

Typically, for a sigmoid neurons network, hidden layers are activated by a *ReLU* function, while for the output layer a sigmoid or softmax function is used depending on the type of output, respectively binary or multi-class.

But how can an ANN learn? When a train data is provided to the network, it tries to predict the results. Initially weights of each neuron are set to a small number, close to 0. Each result obtained from the output layer is then compared with the ground truth value, and a *Cost Function* is computed:

$$C(w, b) = \frac{1}{2n} \sum_x ||y(x) - \hat{y}||^2 \tag{2.17}$$

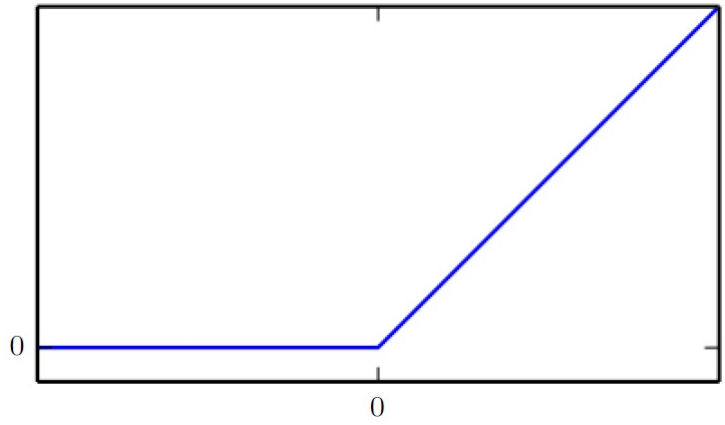


Figure 2.11: ReLU function [2].

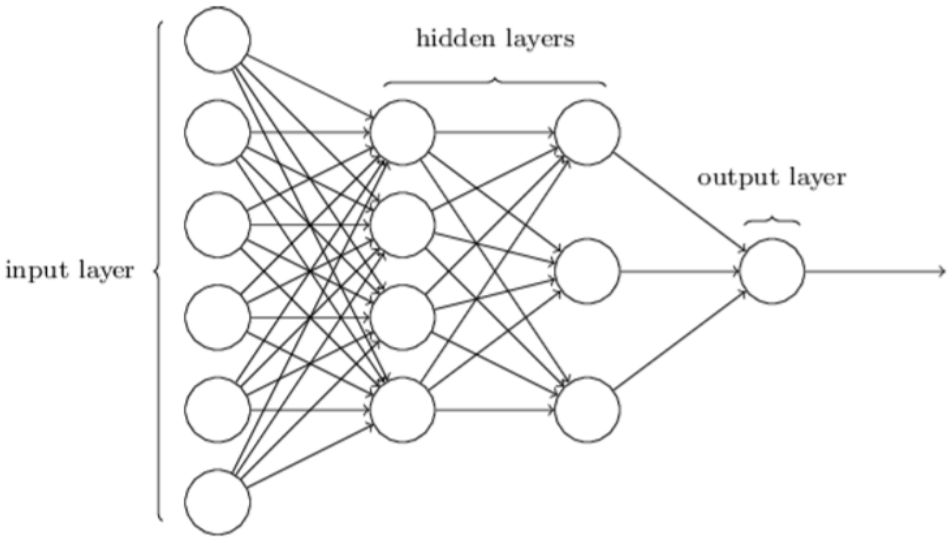


Figure 2.12: Representation of a simple neural network [5].

Where w and b are respectively all the weights and all the biases in the network, n is the total number of training input and $\hat{y}$ is the ground truth value when x is input. This information enters the output layer and proceeds backwards in the network, in an operation called *backpropagation*, at the end of which weights are updated according with how much they are responsible for the error.

The objective of backpropagation (and the main objective of the neural network in general) is to bring the cost function to a null value, which occurs when $y(x) = \hat{y}$. Nearer the cost function to the zero value, more precise will be the network (this is typically attained using a *Gradient descent* algorithm, which will be further explained later). Two different kind of learning technique can be identified:

- **Reinforcement learning**: where weights are updated after each observation;
- **Batch learning**: where weights are updated only after a batch of observation.

2.4.3 Convolutional Neural Networks

In the previous section, fully connected networks were presented, in which every neuron in the network is connected to every neuron in the adjacent layers:

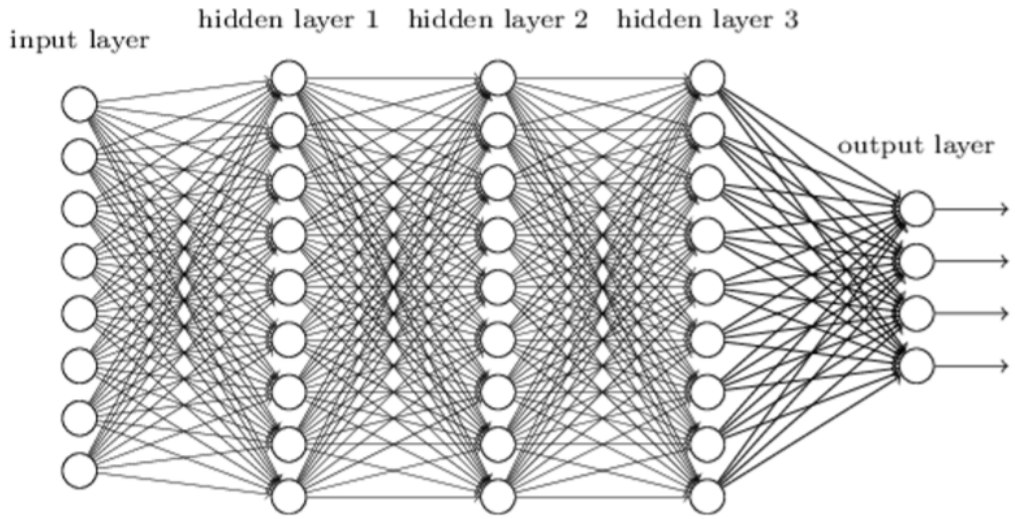


Figure 2.13: Fully connected neural network [5].

In the field of visual imagery, and precisely in that of classification of images, the use of such a network leads to have a input layer with a neuron for each pixel of the image, between which there is no spacial relationship. Indeed the original structure of the image is not maintained, and pixels that are close together or far apart are treated on the same basis. A different kind of network, able to take in consideration the disposition of pixels in the image, would not only start from a pool of a priori information that could improve results, but also make faster and easier the training process.

Convolutional Neural Networks (CNNs) are powerful ANN particularly used in the field of visual imagery, and based on three fundamentals:*local receptive fields, shared weights,* and *pooling.*

local receptive fields Contrary to previous networks, instead of gather input neurons (and so image's pixels) in a vertical column with no relationship between

one and the other, CNN respect the spatial structure of the image. Considering for example a 28 x 28 pixels image, it is convenient to depict the input layer like shown in Figure 2.14.

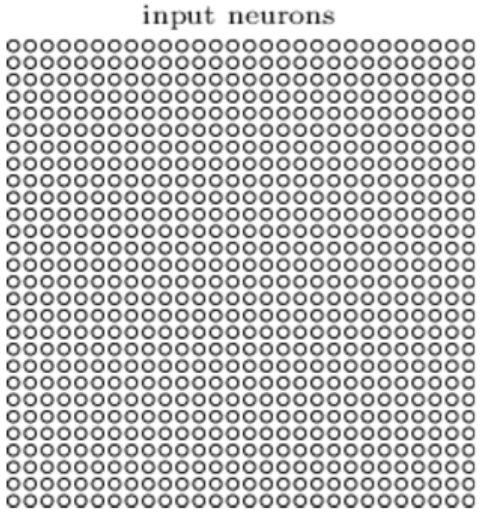


Figure 2.14: CNN input layer [5].

Where each neuron contain the intensity value of each pixel. Then, progressing with the implementation of the first hidden layer, instead of connecting every input pixel with every hidden neuron, only connections in small, localized regions of the input image are performed. As always, each connection will have an associated weight, and each new hidden neuron will have a bias. Sliding the region across the whole image, it is possible to obtain the first hidden layer. Proceeding with the example, 5 x 5 region can be considered for this operation like presented in Figure 2.15.

This whole process is called *Convolution*.

Shared weights and biases At the end of the previous operation a 24 x 24 neurons layer will be obtained. Each of these neuron will have the same weights and bias, which means that all neurons in the first hidden layer will detect the same feature in the input image, but at different locations. This makes CNN well adapted to translation variances, meaning that it will recognize the features in the image also if these are translated in different positions. For this reason, the map from input to hidden layer is called *feature map*, and the weights and bias are called respectively *shared weights* and *shared bias*. Shared weights and bias define a matrix, called *kernel*, *filter* or *Feature detector* (which is the 5 x 5 region previously defined in the example). Anyway, this structure detects just one feature. Generally a convolutional layer is made up of more feature maps, and so different kernel are used. Once the network is trained to recognize a feature,

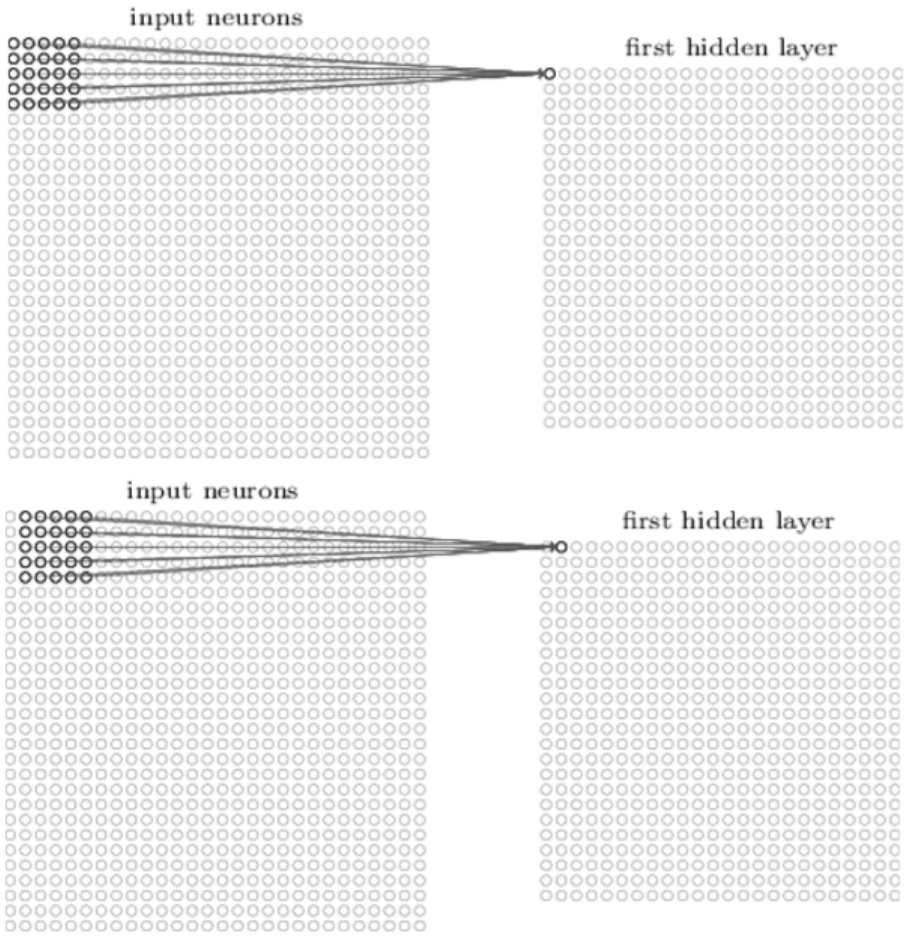


Figure 2.15: Implementation of first hidden layer in a CNN [5].

that particular feature will be detected across the entire image. Remaining in the example, the finale result will be a layer of dimensions x x 24 x 24, where x is the number of feature maps.

Pooling In a CNN, usually immediately after a convolutional layer, a pooling layer is inserted, which simplify the information contained in a convolutional layer. For example, max-pooling, a common pooling procedure, outputs the maximum activation in the 2 x 2 input region.

In such a way the feature is isolated, and a bigger importance is given to the rough location of a feature with respect to the others rather than its precise location; this leads to a drastic reduction in number of parameters needed in later layers, improving the general performance of the network and maintaining the precision of its results. Remaining in the example, the obtained hidden layer after the pooling operation will have a dimension of x x 12 x 12, where x is the number of feature maps from which the pooling operation is performed.

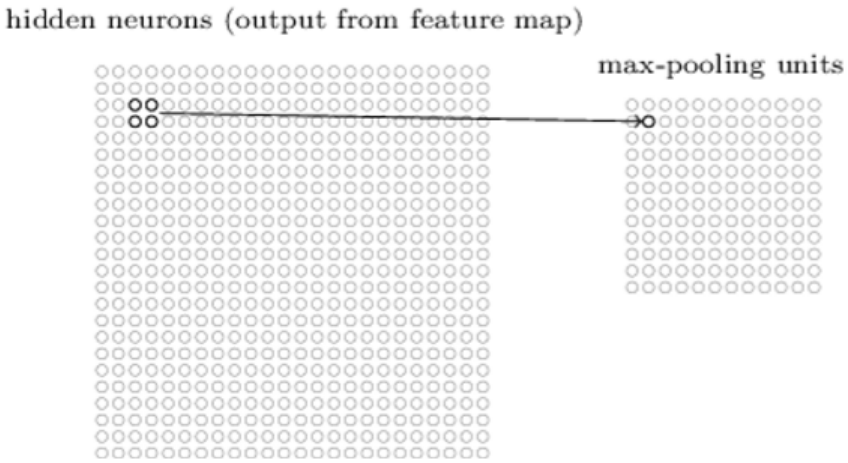


Figure 2.16: Max-pooling procedure in a CNN [5].

Putting it all together, the obtained final neural network presents as input layer a 28 x 28 neuron structure that encode the original image's pixels. This is followed by a first hidden layer of dimensions 3 x 24 x 24, obtained applying a convolutional operation with a 5 x 5 local receptive field and 3 feature maps, and then a second hidden layer of dimension 3 x 12 x 12 obtained applying a max-pooling operation with 2 x 2 regions, across each of the three feature maps. Finally the output layer is made up of X neurons, where X is the number of class contemplated in the problem. This convolutional architecture is quite different to the architectures used in earlier sections, but the overall picture is similar: a network made of many simple units, whose behaviors are determined by their weights and biases. And the overall goal is still the same: to use training data to train the network's weights and biases so that the network does a good job classifying input data.

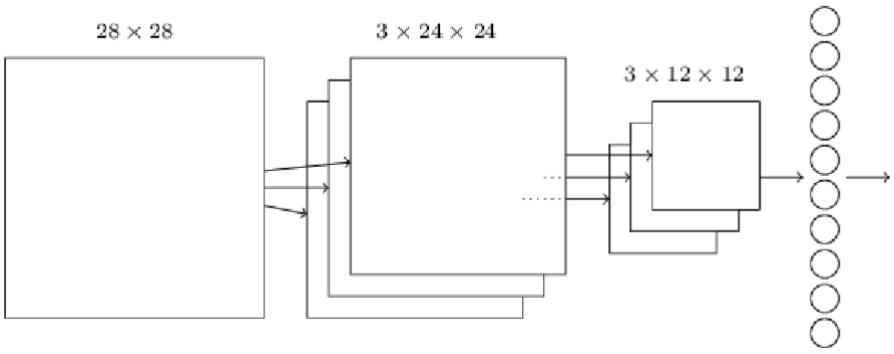


Figure 2.17: CNN architecture [5].

Practice working But how does a Convolutional Neural Network work in practice? As stated before, the first and most important role of CNN is solving of classification problems involving visual imagery. In these problems, the dataset is usually divided in training set and validation set, where the latter is usually a minor fraction of the former (generally the validation set has a number of images equal to a quarter or a fifth of the training set). Then, the CNN model is "fed" with this dataset for a certain number of cycles, called *epochs*. For each epochs, two important operations happen:

- first, the model use the training set to "learn" with a Gradient descent algorithm. This consists in making a prediction for each image of the training set and then compare this result with the ground truth label corresponding to that image, and use the error to update weights and biases;
- then, this model is tested using the validation set. It is not directly responsible for the training of the model, but provide important information to the programmer about its accuracy. Similarly to the training phase, the model try to predict these images one by one, and then this prediction is compared with its ground truth label. The ratio of correct predictions to the total represent the accuracy value of the model for that particular classification problem. It is worth noting that validation set, and consequently its split, is not fundamental for the model to learn, but it is good practice to obtain a quality feedback of the model before applying it.

Often datasets contain a considerable number of images and the system could not permit to maintain all of them in memory, so they are gathered in groups of the same size; each of these groups is called *Batch*. Depending on the batch size, gradient descent algorithm can be replaced by:

- **Batch Gradient Descent**, when the batch size is equal to the whole training set;
- **Stochastic Gradient Descent**, when the batch size is equal to one, which means it is equal to the total number of samples in the training set;
- **Mini-Batch Gradient Descent**, when the batch size is greater than one but smaller than the total number of samples in the training set.

Mini-batch gradient descent seeks to find a balance between the other two implementation, and is the most common algorithm of gradient descent used in deep learning. It is so important because contrary to the simple gradient descent, allows to update weights and biases of the model after the training with each batch. In such a way, considering always that an epoch is counted after the training on the whole training set, and so on all the batches, the model update frequency will be equal to the number of epochs divided by the number of batches. This allows a

more robust convergence, avoiding local minima and improving accuracy, and at the same time lightens the computation effort by avoiding to have all the images always in memory.

Chapter 3

Robot

Before talking about how a robot can navigate an environment and localize itself in it, it is necessary to introduce what it is, in practice. A robot is a machine able of preforming a series of actions autonomously, that means without the direct intervention of an external agent. Robots are generally able to move themselves or to move physical objects, and are mainly used to assist humans, during work or other daily activities, replace theme in the execution of a task, or perform potentially dangerous actions at their place. Robots can have different structure depending on their function:

- Mobile robots are able to move in a specific environment, which can be more or less wide, determined by the function assigned. The branch of mobile robots is very large, ranging from civil use like domestic robots for cleaning and maintenance or autonomous vehicle, to military and space exploration rovers;
- Industrial robots are generally manipulators constituted of a jointed arm with many degree of freedom, ending with a gripper or other different tools. They are usually automatically controlled, reprogrammable and very flexible, able to fulfill many tasks;
- Humanoid robots resemble a human body, with torso, head, two arms and two legs. Like other robots, the design has generally a functional motivation, like experimental purpose on bipedal locomotion or other human tasks, but could also have an aesthetic purpose like, for example, in the field of service robotics for children and elders assistance;
- Bio-inspired robots have a design that attempt to translate biological concepts into engineered systems. These robots are more often inspired rather than copied from nature, which means biological structures are taken and simplified in the robotic design. They can have different use, from natural research to military purpose;

- UAVs, unmanned aerial vehicles, commonly known as drones, are aircraft guided without a pilot on board. The most famous of this kind are quadcopters. Also in this case, they can have different applications, and are used in military, commercial, scientific, agricultural, recreational, surveillance, product delivery and other activities;
- Modular robots are created for multifunctional purpose, and are constituted of different module types, allowing hyper-redundancy with more than eight degree of freedom.

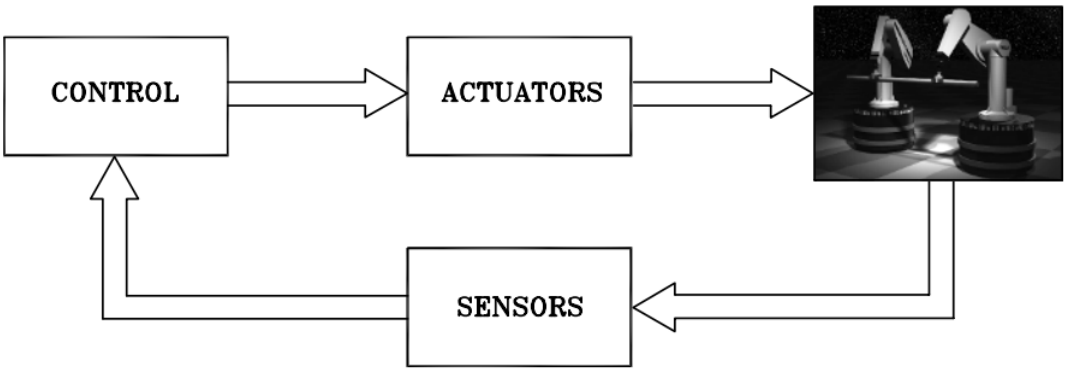


Figure 3.1: General mechanical structure of robots [6].

Robotics is commonly defined as the science studying the *intelligent connection between perception and action* [6]. Figure 3.1 represent the general structure of a robot. The capacity to exert an action, both locomotion and/or manipulation, is provided by an *actuation system* which animate the mechanical components of the robot. This system is generally composed of servomotors, drives, transmissions, and other mechanical components able to obtain the desired functionality. The capability of perception is entrusted to a *sensory system* which can acquire data on the internal status of the mechanical system (*proprioceptive sensors*) as well as on the external status of the environment (*exteroceptive sensors*). The connection between perception and motion is practiced by a *control system* which can command the actuation system on the basis of the information retrieved from the sensory system, and of the task the robot has to complete. The feedback principle followed by the robotic system in this context is very similar to that followed by the human body.

3.1 Sensors

There are a vast number of different sensors being used in robotics, applying different measurement techniques, and using different interfaces to a controller. The scope of this section is not presenting all kinds of robotic sensors but more

specifically those sensors mainly used for SLAM operations. First of all, it is necessary to make a classification based on the kind of application [7]:

- Local or on-board sensors, mounted on the robot;
- Global sensors, mounted outside the robot, in its environment, transmitting data back to the robot;
- Internal or proprioceptive sensors, monitoring the robot's internal state;
- external or exteroceptive sensors, monitoring the robot's environment state.

3.1.1 Accelerometer and Gyroscope

Accelerometer and Gyroscope are generally local and proprioceptive sensors, able to determine the orientation of the robot in a 3D space, which is of primary importance in problems requiring the tracking of the robots, in balancing and walking robots or autonomous planes. More of the same sensor can be integrated in order to measure two or all three axes of orientation.

Accelerometes are electromehcanical device that measure acceleration, the rate of change in velocity of an object, but can not directly measure the orientation of the body, and are often very imprecise and need software filtering to obtain better results.

On the contrary, gyroscopes can measure the orientation of the body with great precision. Combinations of accelerometers and gyroscopes are often used for the creations of advanced electronic devices, like the *Inertial Measurement Unit* (also called IMU), which is able to reports a body's specific force, angular rate and orientation.

3.1.2 Digital Cameras

On mobile robots, digital cameras are usually local and exteroceptive sensors, and a fundamental part of the machine vision system.

Digital cameras are complex sensors because of the processor speed and memory capacity required. Since there is always a trade-off between frame rate and resolution, the former is often preferred upon the latter. In fact, for mobile robot applications, the most important aspect of a camera is an high frame rate, because during the movement it is required to retrieve updated sensor data as fast as possible. Furthermore, the frame rate can be also influenced by the pre- or post-processing process applied on the received image.

On the oter side, the image resolution must be high enough to detect a desired object from a specified distance. This object can be properly recognized only if, on the camera, it is received as a certain number of pixels.

Anyway, in the last years, the compromise between frame rate and resolution has been dwindling, and now cameras with high resolution and frame rate are

commonly available on the market. This gave birth to another problem, that is the passage of much more data, which require additional, faster storage components.

Often, it is more robust to describe robot tasks and environments in three dimensions than in 2D. This because the 3D shapes of tasks and environments are invariant to changes in the scene, like for example lighting, shadows, occlusions and so on. When two cameras are rigidly mounted to a common mechanical structure, they form a *stereo camera*. The two cameras are aligned on the same x-axis and their y- and z-axis are parallel, making possible to determine the depth to points in the scene from the center point of the line between their focal point, similarly to how works the human eye.

Simple stereo cameras are passive devices, namely they passively observe the scene without giving any other contribute, and the real depth detection is entrusted to the software. This lead to a series of complications, and the depth recognition is influenced by a series of factors, both related to the camera (resolution, lens type and quality and so on) and to the environment (lighting, shadows, and so on). Furthermore, this kind of implementation can lead to a very known problem in computer vision, called *correspondence problem*, in which the software has difficulties in recognizing which parts of one image correspond to which part of an other. How it will be understandable in the next chapters, this might be a big problem in SLAM application.

3.1.3 Range Data

The problems discussed previously affecting stereo cameras can be solved by using an active device, able to illuminate the scene in various ways emitting beams and structured light, rather than observe the scene passively. These devices are called *deapth cameras* or *range* cameras.

Depth cameras can operate according to a number of different techniques. One of the most famous and used is *Stereo triangulation*, which work similarly to passive stereo vision system, but is able to partially solve all the problem affecting these cameras. Stereo triangulation introduce an active contribute, which consist in a controlled light emitter, like laser or infrared beam, which project an unstructured light pattern. In this case, as in conventional stereo case, the position of the bright spot where the laser beam strikes the surface of interest is found as the intersection of the beam with the projection ray joining the spot to its image, but contrary to conventional stereo case, the laser spots are much brighter than the other scene points, avoiding the correspondence problem.

An other widespread technique is the *Time-of-Flight* (ToF), which work by measuring the phase-delay of reflected infrared LED or laser illuminator, meaning the time required for these light pulses to fly into the scene and bounce back to the depth camera, capturing a whole scene in three dimensions. This technique is very sensitive to radiometric, geometric and illumination variations, for example measurement accuracy is limited by power of the IR signal, which is usually low

compared to daylight. An other critical problem with ToF is *motion blur*, caused by either camera or object motion. Depth accuracy and frame rate are limited by the required integration time of the depth camera, longer integration time usually allow higher accuracy of depth measurement, which is particularly adapt for static scenes, while for capturing moving objects may be necessary to shorten the integration time, an so the accuracy.

Structured light cameras is another kind of depth camera worth mentioning. It project a precisely known pattern of light into the scene, the camera observe how this pattern is deformed as it lands on objects and surfaces in the scene, and a reconstruction algorithm estimate the 3D structure of the scene from this data.

3.1.4 Laser scanning

Although depth cameras are very precise and efficient in their work, there are still some applications in which *Laser scanners* are widely employed, because of their superior accuracy and longer sensing range.

One of the most famous laser scanner devices is the *LiDAR* (*Light Detection And Ranging*), which transmit an amplitude modulated laser beam to a spot of the 3D surface and receives the reflected signal back. By comparing the change of fase, introduced by the delay, between the transmitted and received signal, the device is able to measure the distance in terms of the period of the modulation of the laser beam. These computations are generally always handled by the firmware of the device.

Laser scanners find several uses in the construction of high-resolution maps with different applications, for laser altimetry tasks, and recently have also been employed in control and navigation for some autonomous cars. In the latter case, laser scanners are significantly different from those used for indoor or slow-moving robots, due to the differences in velocities and distance ranging they have to deal with, but also for aerodynamic forces, vibrations and temperature gaps. To overcome this necessities, on autonomous vehicles LiDARs are often mounted in series, producing multiple scanlines.

Similarly to Time-of-Flight cameras, laser scanners are very sensitive to the different environment conditions, which can change or even fade the laser beams in unexpected ways. For example, LiDAR pulses may be affected by heavy rains, high sun angles or huge reflections, caused for instance by glass windows or water surfaces. Furthermore laser scanners are generally slower compared to depth cameras, due to the time needed to compute the phase change for each spot, and also much more expensive, because of mechanical parts used to steer the laser beam.

3.2 ROS

Until this point, the hardware structure of a robotic system has been presented, neglecting one of the most important parts: *robot control*. The software implementation of robot control has always been one of the most complex and studied topic in robotics. The need for an open collaboration framework was felt by many people in the robotics research community: this lead to the birth of ROS. The *Robot Operating System* (ROS) is a framework for writing robot software, one of the most used all around the world. It is a collection of tools, libraries and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms. Since creating truly robust and general-purpose robot software can reveals very challenging, and it is very difficult and expensive to build a complete system from scratch, ROS was built to encourage collaborative robotics software development. In order to understand the implementations used in this project, a briefly introduction to the main characteristics and functions of ROS is needed.

3.2.1 ROS graph and `roscore`

A ROS system structure can be easily illustrated by a ROS *graph*. It present many different programs running simultaneously and communicating with one another by passing *messages*. These programs are called *nodes*, because they are represented as nodes on the ROS graph, and programs that communicate with one another are connected by *edges*, which represents a stream of messages. Typically, a software crash will only take down its own process (and the corresponding node), while the remaining structure will stay up.

Nodes are able to find one another thanks to `roscore`, a service that provides connection information to nodes, so that they can transmit messages to one another. Every node connects to `roscore` at startup to register details of the message streams it publishes and the streams to which it wishes to subscribe. How it is understandable, `roscore` is a fundamental part of ROS, and no ROS system can exist without it running.

3.2.2 `rosrun` and `roslaunch`

ROS software is organized into *packages*, each of which contains some combination of code, data and documentation. The ROS ecosystem includes thousands of publicly available packages in open repositories. Since these packages are locations in the filesystem with very long paths, it would be tiresome and time-wasting looking for theme one by one. ROS provides a more efficient way to run its programs, a command-line called `rosrun`. The syntax expects the command `rosrun` followed by the name of the package and the name of the executable, and eventually also other parameters compatible with the specific program. Another important

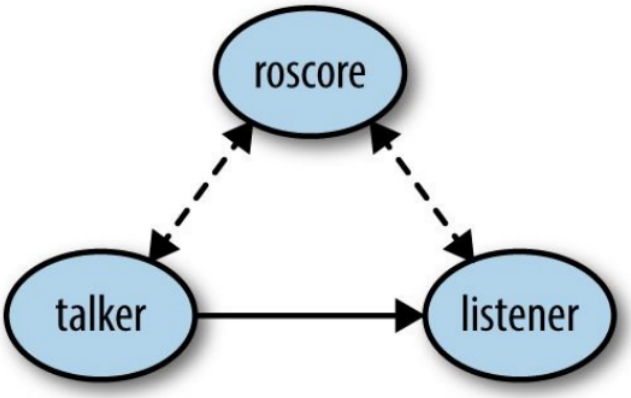


Figure 3.2: `roscore` provide information between two nodes so that they can transmit messages to one another [8].

command-line is `roslaunch`, designed to automate the launching of collections of ROS nodes. The syntax is almost identical to the `rosrun` command-line, expecting the command `roslaunch` followed by the name of the package and the name of the launch file, and eventually other parameters. `roslaunch` is very useful because not only can run a series of nodes, but also other launch files. Furthermore, when a `roslaunch` is closed, all of the nodes associated with it are closed as well.

3.2.3 Topics, Services and Actions

ROS nodes, by themselves, are not very useful until they communicate with each other, exchanging information and data. The most common way to do that is through *topics*, which are names for streams of messages with a defined type. Topics implement a *publish/subscribe* communication mechanism: before starting to transmit data, a node has to *advertise* both the topic name and types of messages that are going to be sent. Then they can start to *publish* the actual data on the topic. Nodes that want to receive messages on a topic can subscribe to that topic by making a request to `roscore` and, from then on, all the messages on the topic will be delivered to these nodes. It is worth noting that all messages on the same topic must be of the same data type.

The publish/subscribe model is a very flexible communication paradigm, but is not appropriate for request/reply interaction. *Services* are another way to pass data between nodes in ROS. They are synchronous remote procedure calls, defined by a pair of messages: one for the request and one for the reply. That allow one node to call a function that executes in another node. Service calls are well suited to things that only need to do occasionally and take a bounded amount of time to complete. They are useful for synchronous request/reply interactions, namely those cases where asynchronous ROS topics don't seem like the best fit.

However, services are not the best fit, either, in those cases when it is needed to initiate long-running tasks. For these situations, ROS provides *actions*, asynchronous procedures which in addition to set a goal and give a result, like services, also use *feedback* to provide updates on the behavior's progress toward the goal and allow to cancel or modify such goal. An action is essentially a higher-level protocol that specifies how a set of topics (goal, result, feedback, etc.) should be used in combination.

3.2.4 Teleoperation

Through nodes, it is possible to make a robot perform all a series of actions and computations in order to accomplish a certain task. But how is it possible to physically move a robot in the environment?

For a terrestrial wheeled robot, the control can be actuated acting just on two velocity degree of freedom: the linear velocity (forward/backward), and the angular velocity around the Z-axis which is, in other words, how fast it is spinning. It is simple to understand how with just these two components it is possible to fully drive a wheeled robot in his 2D environment for all control configurations, whether manual or automatic. ROS provide a function to manually control the movement of the robot through *teleoperation*, using input from a computer keyboard or from a variety of different controllers.

From a practical point of view, the robot is set in communication with the controller source, generally with Bluetooth or through a LAN network, and then the node responsible for the teleoperation (that can usually be the `teleop_twist_keyboad` for general twist robots, or the `turtlebot_teleop` for turtlebot robots) is launched, allowing the control fundamentally through five commands, two for adjust forward and backward velocity, two for both rotation speeds, and another one for immediately interrupt any motion.

3.2.5 `rviz`

`rviz` stands for *ROS visualization*, is a general-purpose 3D visualization environment for robots, sensors and algorithms. A role of paramount importance is taken by the *frame of reference*: when `rviz` is launched to visualize, for example, a controllable robot, every type of data must be associated to a reference frame. This means that, remaining in the example, there must be at lest a reference frame associated to the environment, which usually is set as the *fixed frame*, and a reference frame associated to the robot, which usually is the mobile frame desired to be followed during the motion. Each robot can have several reference frames, like the center of the structure base, the camera, or even wheels in mobile robots, while in a manipulator each joint, including the end effector, has its own reference frame. `rviz` presents a number of panels and plugins that can be configured as needed, and integrates perfectly with the ROS system, allowing to visualize a whole series

of plugins corresponding to topics running in the actual ROS system. In such a way, it is possible to visualize all a series of services corresponding to different parameters of the robot and its sensors, like position and orientation in the environment of each reference frame, trajectory, camera's output images, depth cloud, environment grid map, and so on.

3.3 Hardware components employed

For the purpose of this thesis project, it is necessary a robot able to respect a series of constraints, mainly concerning the robotic structure and the availability of sensors. First of all, it must be small enough to easily roam in complex domestic indoor environments, nimbly avoiding obstacles and impediments, but also big enough to contain all the necessary hardware, including the central processing unit, all without constituting a danger for people or object inside the indoor environment. Furthermore it must support a number of sensor in order to:

- Estimate the 3D structure of the scene, for reconstructing the map of the environment;
- Localize itself in such map;
- Capture some image frame of the environment, which will be used by the deep learning algorithm to classify rooms. For this purpose, the more high is the quality (resolution) of these images, the more precise will be the prediction.

3.3.1 Robotic structure

Terrestrial wheeled robots and UAVs, especially quadcopters, are the two types of robots most largely employed for navigation and SLAM operations. While quadcoperts are used for both outdoor and indoor SLAM operations, terrestrial wheeled robots are simpler, can support more hardware without strict weight limits, and in a domestic scenario are much safer. The choice therefore fell on *TurtleBot3*, in particular the *Waffle* version.

TurtleBot is a minimalist platform for ROS-based mobile robot education and prototyping, developed by the Korean manufacter ROBOTIS [10]. It present a small differential-drive mobile base with an internal battery, power regulators and changing contacts, and is constituted of a stack of laser-cut "shelves", covered with mounting holes, allowing to customization by adding additional hardware subsystems, like manipulators, sensors, upgraded computers, and so on. By default, the fundamental components provided with the TurtleBot3 Waffle are:

- An Intel Joule 570x Single Board Computer, an embedded computer controller complete of 16GB of storage, 4GB of included memory and an integrated graphic processing unit, all with a system running on Linux;

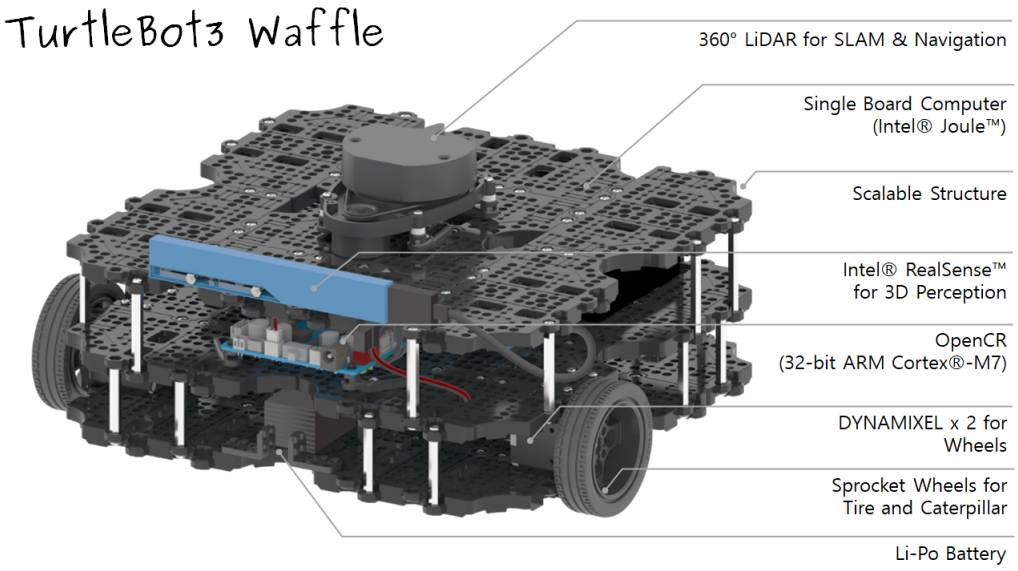


Figure 3.3: Principal TurtleBot3 Waffle components [9].

- Two Dynamixels, high-performance, all-in-one actuators which control the two front wheels;
- An OpenCr (Open-source Control Module for ROS), developed for ROS embedded systems, which mount a STM32F7 series chip based on a very powerful 32-bit ARM Cortex-M7 microcontroller with floating point unit. This whole system is used to the control of sensors and actuators;
- A 360 LiDAR sensor;
- An Intel Realsense R200 active stereo depth camera for range sensing;
- An inertial measurement unit constituted of a 3 axis gyroscope, a 3 axis accelerometer and a 3 axis magnetometer;
- A Lithium polymer 11.1V 1800mAh/19.98Wh battery.

For this project, the Intel Joule 570x is replaced with a more powerful NVIDIA Jetson AGX Xavier Developer Kit unit [11], which present a 512-core NVIDIA Volta GPU, an 8-core ARM 64-bit NVIDIA XAVIER CPU, 32GB of flash storage and 32GB of DRAM memory, with an operative system called JetPack, a *Software Development Kit* (SDK) based on Linux as well [12]. Moreover, in addition to the DC power socket, it is provided with an HDMI port for system installation and implementation, an Ethernet port for internet connection, two USB type C and one USB 3.1 type A ports, allowing connection of many components.

Figure 3.4: Jetson AGX Xavier [11].

The Jetson AGX Xavier was mainly designed for robots and autonomous machines, so it is particularly adapt for real-time execution and is able to perform neural networks computation very efficiently, thanks to integrated software libraries as CUDA, cuDNN and TensorRT. Because of its considerable size, it has been necessary to slightly modify the TurtleBot3 Waffle in order to make enough space to contain it.

3.3.2 Employed Sensors

In addition to TurtleBot3 Waffle's integrated sensors, an other pair of cameras have been added to the structure, to allow a more precise and robust SLAM execution, and to retrieve better RGB images for the deep learning classification algorithm.

Intel RealSense D435i Depth Camera An active IR stereo depth camera used to replace the Intel RealSense R200, which potentially has all the characteristics needed by this project. In fact, Intel RealSense D435i is provided with two infrared cameras and an infrared projector. The two IR cameras sees both IR and visible light, performing dense stereo matching based on feature it can see in visible wavelength, while in infrared they pick up also on artificial features generated by the projector [13]. The operating range is between 0.105 and 10 meters at 86x57 degrees of FOV, with a depth resolution of 1280x720 at up to 90 fps. This solutions, united with the integrated Intel RealSense Vision Processor D4, which

takes care of rectifying and processing the images captured by cameras, allows a very precise and robust 3D reconstruction of the environment.

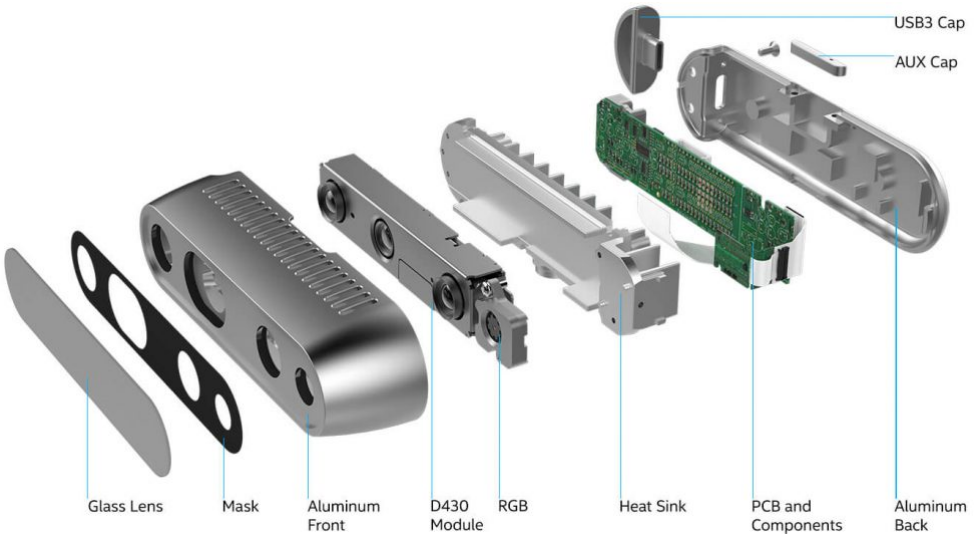


Figure 3.5: Intel RealSense D435i structure [14].

D435i mounts also Full HD RGB camera at 30 fps, with a 69.4x42.5x77 degree of FOV, that is perfect for the acquisition of frames for the deep learning classification algorithm.

Finally, Realsense D435i has also an IMU unit, which combine a variety of sensors with gyroscopes to detect both rotation and movement in 3 axes, as well as pitch, yaw and roll. In such a way, knowing the initial pose of the robot, it is possible to track its subsequent positions and orientations with respect to the environment in any moment.

Even thought this camera seems to be the perfect fit for this project, and seems to have all the characteristics needed to achieve the prefixed goals, the localization offered by its IMU unit in many cases results not very robust, leading to lost of tracking, even with the slightest jolt, without any possibility of recovery. This lead to the search of a new device able to compensate this weakness.

Intel RealSense T265 Tracking Camera A tracking camera is a camera specifically designed for localization and pose estimation. This particular Intel model is provided with two OV9282 Fisheye cameras that transmit monochrome images at a risolution of 484x800 and 30 fps, with a field of view of 173 degree each, in addition to an inertial measurement unit constituted of a series of Accelerometers and Gyroscopes in a single package. For processing of the images retrieved by cameras, the T265 use a VPU (*visual processing unit*) Movidius MA215x.

For localization and pose estimation, Intel RealSense T265 exploit its advanced inertial measurement unit, similarly to D435i, but contrary to this it integrate this

Figure 3.6: Intel RealSense t265 [15].

IMU with a *Virtual Inertial Odometry* (VIO) system, which determine position and orientation of the robot by analyzing the associated camera images, with an appearance-based loop closure detection (explained in the following chapter).

The integration between Intel RealSense D435i and T265 allow a more robust and reliable localization and tracking of the robot, leading to a better pose estimation and improving the precision in building the 3D map of the environment. Actually, the joint use of these two camera is employed, and also very encouraged by Intel for SLAM projects [16].

Turtlebot Lidar The lidar module mounted on the TurtleBot3 Waffle is a 360 Laser Distance Sensor LDS-01, a 2D laser scanner capable of sensing 360 degrees that collects a set of data around the robot.

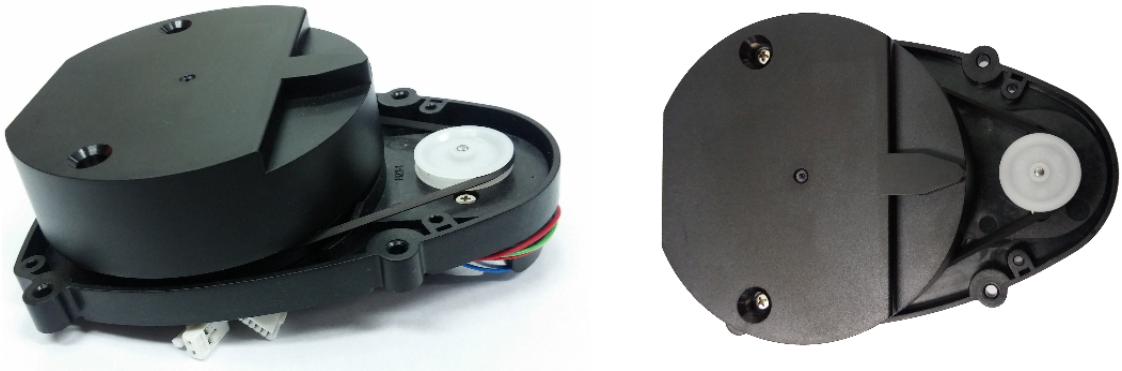


Figure 3.7: 360 Laser Distance Sensor LDS-01 [17].

How will be discussed in next chapters, the depth camera can be used instead of the lidar, on the other hand the latter is a relatively simple and cheap device that allows to obtain a better quality map of the environment.

Chapter 4

SLAM

4.1 Introduction

The simultaneous localization and mapping is the computational problem of constructing the map of an unknown environment with a robot while simultaneously localizing and tracking the position of said robot, in relation to this map. In the field of robotics, it is historically known as a fair complex problem: to localize a robot, *a priori* information about the environment's map are necessary, while to construct said map, it is necessary to know the pose of that robot in any moment. When both these information are unknown the problem gets exponentially complicated, but nowadays several technique to solve it exist. Formerly, it is possible to recognize two kind of SLAM:

- **Online SLAM**: involves estimating the posterior over the momentary pose along with the map.

$$p(x_t, m | z_{1:t}, u_{1:t}) \tag{4.1}$$

  where x_t is the pose at time t, m is the map, z and u are respectively the measurements and controls.

- **Full SLAM**: involves calculating a posterior over the entire path $x_{1:t}$ along with the map, instead of just the current pose x_t.

$$p(x_{1:t}, m | z_{1:t}, u_{1:t}) \tag{4.2}$$

Furthermore, SLAM problem is continuous and discrete at the same time. The localization of object in the map and the robot's own pose constitutes a continuous problem, while correspondence of an object, as a landmark, with an other previously detected, is typically a discrete problem. This duality is manageable thanks to some algorithms (like EKF, discussed later at section 4.5.2), able to estimate

both the robot pose, which is continuous, and the correspondences of measurements and landmarks in the map, which are discrete. The successive sections will be organized like follows:

- first of all, in order to better understand next arguments, Section 4.2 will briefly explain the concepts of Bayes filter and Markov assumption, which are the basis for almost every SLAM algorithms;
- then, Sections 4.3 and 4.4 will expose the two basic operations which constitute the SLAM problem: Localization and Mapping;
- finally, from Section 4.5 to 4.8, a series of the most popular filter and associated technique used for solving SLAM problems will be presented.

4.2 The Bayes filter

At the base of the Bayes filter there is the concept of *Probabilistic Robotics*. The key idea is to represent the uncertainty as the calculus of probability, that means instead of relying on a single "best guess", represent information by probability distributions over a space of possible hypotheses. In this context, the probability function describing the robot's momentary estimate of its state (or *pose*) is referred to as *belief*.
The Bayes filter, already introduced in (2.15), can be furthermore appreciated with this equation:

1:	**Algorithm Bayes_filter(**$bel(x_{t-1}), u_t, z_t$**):**
2:	for all x_t do
3:	$\overline{bel}(x_t) = \int p(x_t \mid u_t, x_{t-1})\ bel(x_{t-1})\ dx$
4:	$bel(x_t) = \eta\ p(z_t \mid x_t)\ \overline{bel}(x_t)$
5:	endfor
6:	return $bel(x_t)$

Table 4.1: General Bayes filter algorithm [18].

In order to retrieve the belief of the state x at time t, $bel(x_t)$, the Bayes filter take as input the belief at time $t-1$, along with the most recent control u_t and measurement z_t. This operation is called *update rule* and is applied recursively to calculate the belief $bel(x_t)$ from the belief $bel(x_{t-1})$ for every t. In particular the update rule is accomplished through two major steps:

- the **prediction**, depicted in line 3, where the belief $\overline{bel}(x_t)$, that is the one assigned by the robot to the state x_t, is calculated by the integral of the

product between the belief $bel(x_{t-1})$ and the probability that the control u_t induces a transition from x_{t-1} to x_t

- the **measurement update**, depicted in line 4, where the previously obtained belief $\overline{bel}(x_t)$ is multiplied by the probability that the measurement z_t may have been observed in correspondence to the state x_t, and normalized through the constant η, to finally obtain the belief $bel(x_t)$.

In conjunction with the Bayes filter, Markov assumption is of fundamental importance. It states that past and future data are independent if one knows the current state x_t, in other words the trajectory of the robot can be neglected as long as we know its current pose.

4.3 The localization problem

For mobile robots, localization is the problem of determining the pose of the robot relative to a given map of the environment. It is at the base of almost every problem in robotics, simply because all tasks require knowledge of the location of the robot and objects around it to perform an action. Localization can be seen as a problem of coordinate transform. It is necessary, in fact, to detect and monitor the reference frame integral with the robot in relation with the reference frame of the environment and objects contained therein. Knowing the pose $x_t = (x, y, \theta)^T$ is usually sufficient for such coordinate transform, but this pose generally can not be sensed directly by the robot and must be retrieved through other means. Not only, usually a single localization method and a single measurement are not sufficient to guarantee a precise pose estimation, instead a set of localization technique are needed, and the robot has to integrate data over time to determinate a pose.

Localization problems can be more or less difficult depending on a certain number of variables, as the initial knowledge of the robot in relation with the environment, the characteristic of the environment itself and the type of approach used to solve the problem.

Initial information of the robot

- **Position tracking**: the initial robot's pose is known, and the noise for the subsequent poses is assumed to be small. This is a *local* problem, since the uncertainty is local and confined to a region near the robot's true pose;
- **Global localization**: the initial robot's pose is unknown, so it can be everywhere in the environment;
- **Kidnapped robot problem**: this is a variant of the global localization problem, where the robot can be taken and teleported to some other location

during the operation. This problem can be very complex since while in the global localization problem the robot know that it does not know where it is, here it can think to know where it is when it does not.

Environment characteristic

- **Static environment**: the only variable quantity is the robot's pose, everything else that characterize the environment is fixed and can not change position for the whole duration of the robot's operation;
- **Dynamic environment**: both, the robot pose and parts of the environment can change position over time, during the robot's operation. Most real environments are dynamic. Obviously, localization in such environments is much more difficult, just think, for example, that the robot may take a moving object as landmark.

Passive and Active approaches

- **Passive localization**: the localization module observe the robot operating but can not, in any case, take control of the robot or influence its motion;
- **Active localization**: the localization module control the motion of the robot to facilitate the tracking operation and minimizing the localization error. This approach generally leads to better results, since instead of waiting for the robot to move in a favorable location, the localization algorithm can guide it there exactly when and how it needs it.

4.4 The Mapping problem

In the robotics field the *Mapping problem* is intended as the problem of constructing a map of the environment in which a certain task has to be actuated. This map does not have to be perfectly faithful to the real world, but accurate enough depending on the kind of operation required.

In general, a map of the environment is a list of objects and their locations, and can be usually indexed in one of two ways:

- a **feature-based** map is a list of features. Each feature correspond to a distinct object in the real world, and posses the information of its location, in robotics it is common to call these object *landmarks*;
- a **location-based** map is a list of locations, that are generally referred to with coordinates.

With feature-based map, the robot's sensors can measure the range and the bearing of the landmark relative to the robot's local coordinates, then use these information in order to localize itself and to detect new landmarks. In such a way the robot construct a map that does not contain the information about the whole environment, but only about a portion of area in correspondence to most salient elements (landmarks) and positions with respect to them and to himself. If this is an advantage from a certain point of view, in fact the final result will be very compact and lightweight, on the other side does not provide any useful information for navigation. On the contrary, location-based map are volumetric, in that they offer information not only about object in the environment but also about the "free-space", or absence of obstacles.

A classical location-based map representation is known as *Occupancy Grid Map*, where the map is a fine-grained grid defined over the continuous space of locations. The posterior of the map can be written as

$$p(m|z_{1:t}, x_{1:t}) \tag{4.3}$$

where m is the map, $z_{1:t}$ and $x_{1:t}$ are respectively all the measurements and all the poses up to time t. Here, we can define the map as:

$$m = \sum_i m_i \tag{4.4}$$

where m_i refers to the grid cell i. During the mapping process, grid cells not yet analyzed do not contain any value, but when they are approached by the field of view of robot's sensors a binary value is assigned: "0" for free and "1" for occupied. The probability for a cell to be free or occupied will be written $p(m_i)$, and then the problem (4.3) can be rewritten:

$$p(m_i|z_{1:t}, x_{1:t}) \tag{4.5}$$

This problem is simpler because the estimate is performed only over one cell at a time, and not over the whole map.

4.5 Gaussian filters

4.5.1 The Kalman fiter

Historically the earliest and most influent SLAM algorithm is the one based on the Kalman filter (and on the more advanced Extended Kalman filter). It is part of the family of Gaussian filters, which are based on the idea that belief are represented by multivariate normal distributions, depictable as Gaussian functions.

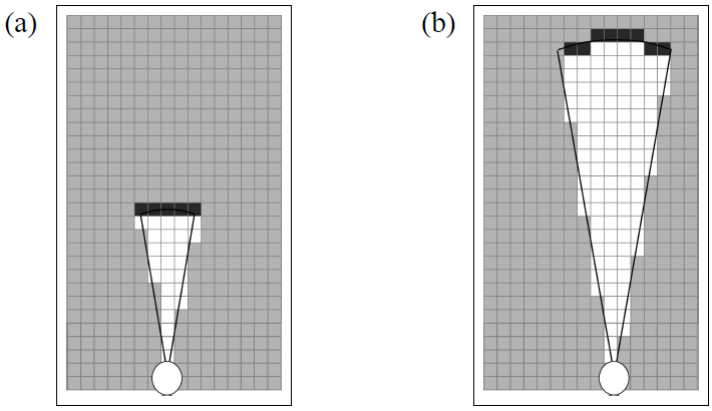


Figure 4.1: Two example of occupancy grid map with different measurement ranges. Grey cells are not yet been examined, white cells indicate free-space, while black cells indicate obstacles [18].

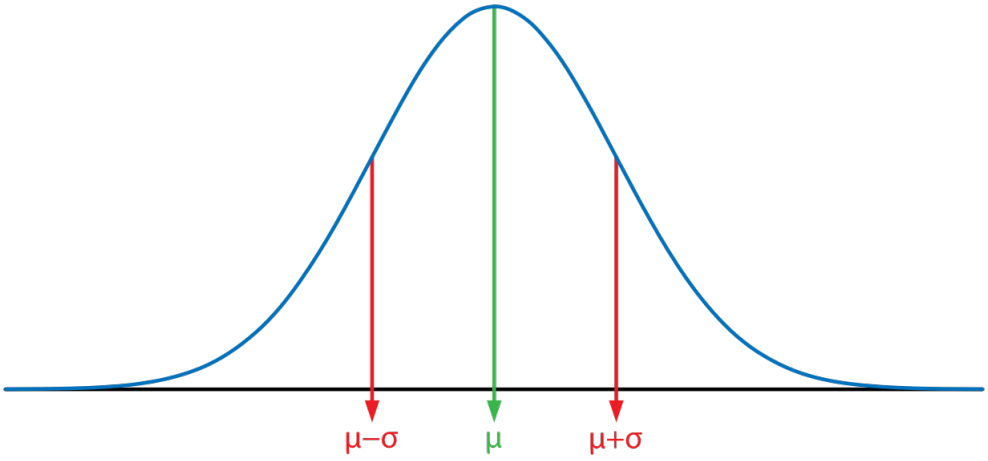


Figure 4.2: General Gaussian function representation [19].

In a Gaussian function the density over the variable x is characterized by two set of parameters: the mean μ and the covariance Σ.

The Kalman filter was invented in the 1950s by Rudolph Emil Kalman, as a technique for filtering and prediction in linear systems, and is the best studied technique for implementing Bayes filters. Its algorithm can be formalized as in Table 4.2.

At time t, the belief of the state x_t, $bel(x_t)$, is represented by the mean μ_t and the covariance Σ_t, taking as input the belief of the state x_{t-1} represented by the mean and the covariance at time $t-1$, and the most recent control and measurement. Then:

1: **Algorithm Kalman_filter($\mu_{t-1}, \Sigma_{t-1}, u_t, z_t$):**
2: $\bar{\mu}_t = A_t\ \mu_{t-1} + B_t\ u_t$
3: $\bar{\Sigma}_t = A_t\ \Sigma_{t-1}\ A_t^T + R_t$
4: $K_t = \bar{\Sigma}_t\ C_t^T (C_t\ \bar{\Sigma}_t\ C_t^T + Q_t)^{-1}$
5: $\mu_t = \bar{\mu}_t + K_t(z_t - C_t\ \bar{\mu}_t)$
6: $\Sigma_t = (I - K_t\ C_t)\ \bar{\Sigma}_t$
7: return μ_t, Σ_t

Table 4.2: General Kalman filter algorithm [18].

- at line 2, the mean of the posterior state $\overline{\mu}_t$ is obtained by the sum of the mean at time $t-1$ and the most recent control respectively linearized by the matrices A_t and B_t. A_t has size $n \times n$, B_t is $n \times m$ where n is the dimension of the state vector x_t and m is the dimension of the control vector u_t;

- at line 3, similarly, the covariance of the posterior state $\overline{\Sigma}_t$ is obtained the sum of the covariance at time $t-1$, linearized with the product between the matrix A_t, and R_t, which is the covariance of the Gaussian random vector ϵ_t;

- at line 4, the *Kalman gain* K_t is computed by the division between the covariance of the posterior state, linearized by the product with the matrix C_t, and the sum between the same linearized posterior state's covariance and Q_t. C_t is a matrix of size $k \times n$ where k is the dimension of the measurement vector z_t, while Q_t is the covariance of the measurement noise vector δ_t. In general, K_t specifies the degree to which the measurement is incorporated into the new state estimate.

- finally at line 5 and 6 the new mean and the covariance of the state x_t are computed through the values obtained previously, plus an identity matrix I.

The Kalman filter is computationally quite efficient, but how does it work from a practical point of view? How illustrated in Figure 4.3, in the *prediction phase* at point (a), for every new pose of the robot an initial belief is stated (generally based om odometry information). Then the *update phase* begin at point (b), where a measurement (in bold) is taken through sensors. Since the initial belief is only a first estimate and the measurement can be affected by noises and uncertainties, both are not singularly sufficiently reliable to establish a reasonably probable pose of the robot. To achieve a solution, the Kalman algorithm is applied and, as evident at point (c), it integrate the two distributions in order to obtain a single, more reliable, result. from point (d) to (f) the same operation is applied to a new position of the robot, to the right of the precedent one.

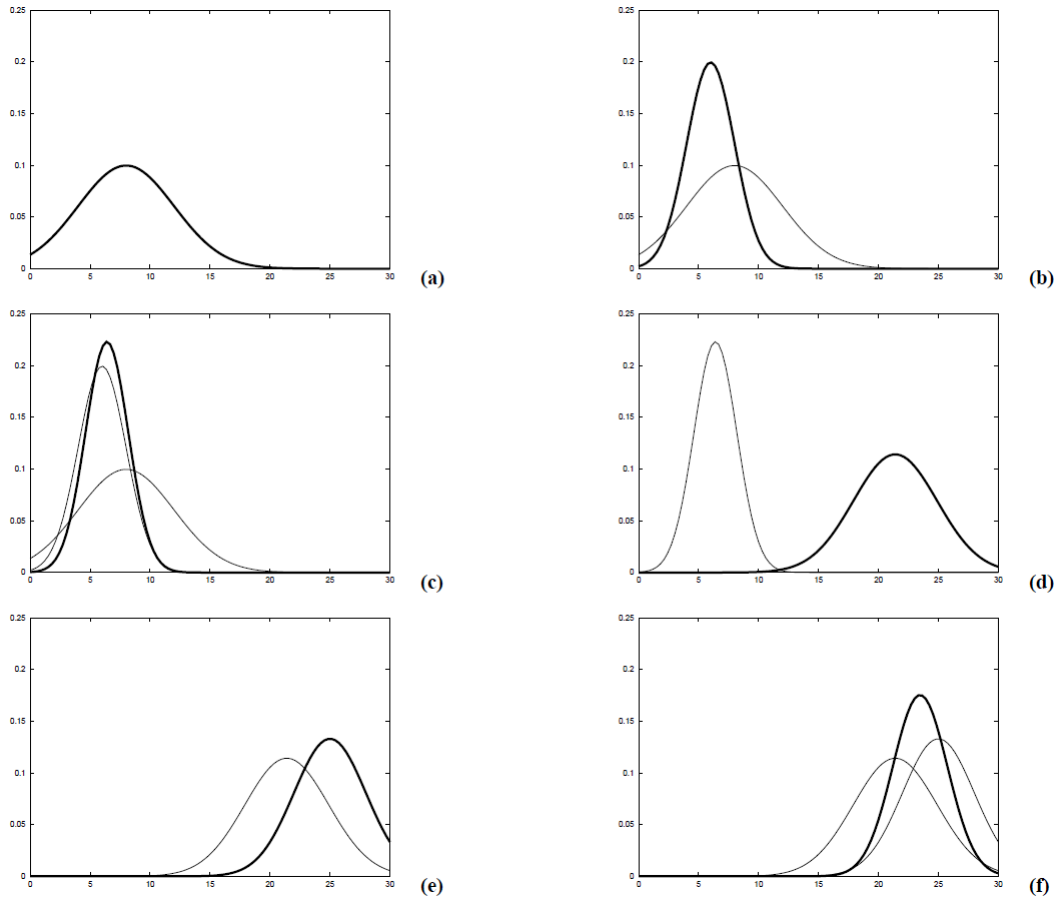


Figure 4.3: Kalman filter functioning [18].

4.5.2 Extended Kalman filter

One of the biggest limits of the Kalman filter is its dependence to the linearity of the system. Since most of practical robotic tasks, including also SLAM operation, are often nonlinear problems, this make the Kalman filter not reliable. A very straightforward solution to this is the *Extended Kalman Filter*. Often referred to as EKF, it is a relaxation of the Kalman filter, which overcome the linearity assumption. This is usually obtained by applying a first order *Taylor expansion*, which is able to approximate the nonlinear function to a linear one.

The EKF algorithm, exposed in Table 4.3, is very similar to the Kalman filter algorithm, the major difference are exposed in Table 4.4.

That means the linear predictions A_t, B_t are replaced by the nonlinear generalization g and similarly C_t by h, and more precisely the Jacobian G_t correspond to the matrices A_t and B_t, while the Jacobian H_t correspond to the matrix C_t.

The EKF is a powerful and very computational efficient algorithm, but has an

1: **Algorithm Extended_Kalman_filter**($\mu_{t-1}, \Sigma_{t-1}, u_t, z_t$):
2: $\bar{\mu}_t = g(u_t, \mu_{t-1})$
3: $\bar{\Sigma}_t = G_t \, \Sigma_{t-1} \, G_t^T + R_t$
4: $K_t = \bar{\Sigma}_t \, H_t^T (H_t \, \bar{\Sigma}_t \, H_t^T + Q_t)^{-1}$
5: $\mu_t = \bar{\mu}_t + K_t(z_t - h(\bar{\mu}_t))$
6: $\Sigma_t = (I - K_t \, H_t) \, \bar{\Sigma}_t$
7: return μ_t, Σ_t

Table 4.3: Extended Kalman filter algorithm [18].

	Kalman filter	EKF
state prediction (line 2)	$A_t \mu_{t-1} + B_t u_t$	$g(u_t, \mu_{t-1})$
measurement prediction (line 5)	$C_t \overline{\mu_t}$	$h(\overline{\mu_t})$

Table 4.4: Main differences between KF and EKF

important limitation, that derive from the use of Taylor expansion for the linear approximation. The goodness of this approximation depends on the degree of non-linearity and the degree of uncertainty of the function. The former because the closer the nonlinear function is to be linear, the better will be its approximation. As a consequence, very nonlinear functions will lead to very unreliable approximations. The latter because the less certain the robot is of its state estimate, the wider will be its Gaussian belief, and so more affected by nonlinearities in the state transition and measurement functions.

In order to overcome such limits, sometimes other approximation methods are preferable. One of these is the *Unscented transform*, which probes the function to be linearized at selected points and calculates a linearized approximation based on the outcomes of these probes. When this transform is used, the filter take often the name of *Unscented Kalman Filter* (or *UKF*). Another one is called *moments matching*, which during linearization preserves the mean and the covariance of the original distribution.

4.5.3 Information filter

Like Kalman filter, the *Information Filter* represents the belief by a Gaussian, but with a substantial difference: While Kalman filters (and as a consequence EKFs) represent Gaussians by their moments (mean and covariance), information filters represent Gaussians in their canonical representation, that means through

an information matrix and an information vector. Such difference leads to have situation in which what is computationally complex in one representation happens to be simple in the other, and vice versa.

The canonical representation of a multivariate Gaussian is given by an information matrix Ω and an information vector ξ, computed respectively as the inverse of the covariance matrix and as the product between the inverse of the covariance matrix and the mean.

$$\begin{aligned} \Omega &= \Sigma^{-1} \\ \xi &= \Sigma^{-1}\mu \end{aligned} \tag{4.6}$$

The Information filter algorithm, as evident from Table 4.5, is almost identical to the Kalman filter algorithm. Matrices A_t, B_t, C_t and Q_t are the same explained in the precedent sections, while the information vector ξ_t/ξ_{t-1} and the information matrix Ω_t/Ω_{t-1} take respectively the place of the mean μ_t/μ_{t-1} and the covariance Σ_t/Σ_{t-1}, following the equation (4.6).

1: **Algorithm Information_filter**($\xi_{t-1}, \Omega_{t-1}, u_t, z_t$)**:**
2: $\bar{\Omega}_t = (A_t\ \Omega_{t-1}^{-1}\ A_t^T + R_t)^{-1}$
3: $\bar{\xi}_t = \bar{\Omega}_t(A_t\ \Omega_{t-1}^{-1}\ \xi_{t-1} + B_t\ u_t)$
4: $\Omega_t = C_t^T\ Q_t^{-1}\ C_t + \bar{\Omega}_t$
5: $\xi_t = C_t^T\ Q_t^{-1}\ z_t + \bar{\xi}_t$
6: return ξ_t, Ω_t

Table 4.5: Information filter algorithm [18].

Just like the Kalman filter, also the Information filter has a relaxation version of its assumptions, called similarly *Extended Information Filter* (EIF).
The analogies between the Kalman and the information filter end their respective extended versions can also be noted in the EIF algorithm, depicted in Table 4.6.

Also in this case, the nonlinear generalizations g and h (and their Jacobian G_t and H_t) replace the linear parameters A_t, B_t and C_t. Since both g and h require a state as an input, Ω and ξ are not sufficient and the computation of μ, at line 2, is required.

This additional step is one of the major throwback of the Information filter over the Kalman filter, and for this it is largely considered to be computationally inferior to his counterpart, the EKF.
On the other size, representing global uncertainty is simpler in the Information filter, and often is numerically more stable. Another advantage of the Information

1: **Algorithm Extended_information_filter**($\xi_{t-1}, \Omega_{t-1}, u_t, z_t$)**:**
2: $\mu_{t-1} = \Omega_{t-1}^{-1}\ \xi_{t-1}$
3: $\bar{\Omega}_t = (G_t\ \Omega_{t-1}^{-1}\ G_t^T + R_t)^{-1}$
4: $\bar{\xi}_t = \bar{\Omega}_t\ g(u_t, \mu_{t-1})$
5: $\bar{\mu}_t = g(u_t, \mu_{t-1})$
6: $\Omega_t = \bar{\Omega}_t + H_t^T\ Q_t^{-1}\ H_t$
7: $\xi_t = \bar{\xi}_t + H_t^T\ Q_t^{-1}\ [z_t - h(\bar{\mu}_t) - H_t\ \bar{\mu}_t]$
8: return ξ_t, Ω_t

Table 4.6: Extended Information filter algorithm [18].

filter over the Kalman filter arise for multi-robot problems, where the integration of sensor dafa collected decentrally, and commonly performed through Bayes rule, becomes an addition. This greatly simplify the numerical complexity of the problem and make the Information filter more flexible than Kalman filter.

4.6 Particle filter

Particle Filter is a nonparametric implementation of the Bayes filter, which approximate the posterior by a finite number of parameters. Instead of representing the distribution of states with a parametric form, as a Gaussian, particle filter represent a distribution by a set of weighted samples drawn from this distribution. These samples are called particles, and are denoted as:

$$\chi_t = x_t^{[1]}, x_t^{[2]}, ..., x_t^{[M]} \tag{4.7}$$

Just like Bayes filter, and how can be seen from Table 4.7, the set of particles χ_t is constructed recursively from the set χ_{t-1}, and need as input also the most recent control u_t and measurement z_t. Then, the algorithm construct a temporary particle set $\overline{\chi_t}$ of dimension M, which is similar to the belief $\overline{bel}_{x_t}$, by generating (at line 4) an hypothetical state $x_t^{[m]}$ based on the particle $x_{t-1}^{[m]}$ and the control u_t. Then, at line 5, a weight $w_t^{[m]}$ is assigned to the particle $x_t^{[m]}$ from the most recent measurement information. From line 8 to 11 the *resampling* occurs, where the algorithm draws with replacement M particles from the temporary set $\overline{\chi_t}$, based on the importance weight of each particle.

A popular localization algorithm based on particle filter is the *Monte Carlo Localization* algorithm. It is applicable to both, local and global localization problems, is easy to implement and has often good performances. Its algorithm is almost identical to the that of the general particle filter, and can be exposed in three phases:

1: **Algorithm Particle_filter**($\mathcal{X}_{t-1}, u_t, z_t$)**:**
2: $\bar{\mathcal{X}}_t = \mathcal{X}_t = \emptyset$
3: for $m = 1$ to M do
4: sample $x_t^{[m]} \sim p(x_t \mid u_t, x_{t-1}^{[m]})$
5: $w_t^{[m]} = p(z_t \mid x_t^{[m]})$
6: $\bar{\mathcal{X}}_t = \bar{\mathcal{X}}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$
7: endfor
8: for $m = 1$ to M do
9: draw i with probability $\propto w_t^{[i]}$
10: add $x_t^{[i]}$ to $\mathcal{X}_t$
11: endfor
12: return $\mathcal{X}_t$

Table 4.7: Particle filter algorithm [18].

- **Prediction phase**, in which a new set of temporary particles $\overline{\chi_t}$ is determined starting from the previous set χ_{t-1} and the most recent control u_t (and sometimes also odometry information);
- **Update phase**, in which each particle is assigned a weight based on the information extracted through sensors;
- **Resempling phase**, in which a new set of particles χ_t, that better approximates the probable robot state, is determined from the information gathered previously.

The precision of MC localization, and in general of the particle filter, is based on the dimension of the particle set M, higher this value, more precise the prediction.

4.7 Histogram filter

The *Discrete Bayes Filter* is applied to problems with finite state space, that means where the random variable X_t can assume different values. Table 4.8 provide its algorithm, and as evident it derives from the general Bayes filter of Table 4.1. x_i and x_k are finite individual states, while $p_{k,t}$ is the probability of each state x_k. At line 3, the prediction (belief) for the new state is calculated, based on the control alone. This prediction is then updated in line 4, to incorporate the measurement.

The greater limit of Discrete Bayes Filter is that it is only useful for discrete systems. There is a variant of its algorithm, used also for continuous systems, called *Histogram Filter*. Histogram filter decompose the state space into finitely

1: **Algorithm Discrete_Bayes_filter($\{p_{k,t-1}\}, u_t, z_t$):**
2: for all k do
3: $\bar{p}_{k,t} = \sum_i p(X_t = x_k \mid u_t, X_{t-1} = x_i)\ p_{i,t-1}$
4: $p_{k,t} = \eta\ p(z_t \mid X_t = x_k)\ \bar{p}_{k,t}$
5: endfor
6: return $\{p_{k,t}\}$

Table 4.8: Discrete Bayes Filter algorithm [18].

many regions, and represent the cumulative posterior for each region by a single probability value.

$$\text{range}(X_t) = x_{1,t} \cup x_{2,t} \cup ... \cup x_{k,t} \tag{4.8}$$

X_t is a random variable describing the state of the robot at at time t, and the function range(X_t) is the state space, that is the set of possible values that X_t might assume. Each $x_{k,t}$ describes a convex region, and together form a partitioning of the state space. A decomposition of a continuous state space is a multi-dimensional grid where each $x_{k,t}$ is a grid cell, and for each cell discrete Bayes filter assign a probability $p_{k,t}$

This concept is at the base of the *Grid localization* algorithm, one of the most popular localization algorithm, which exploits the histogram filter in order to approximate the belief of the robot's state over a grid decomposition of the pose space. The belief $bel(x_t)$ is defined as:

$$bel(x_t) = \{p_{k,t}\} \tag{4.9}$$

and is conceptually almost identical to what seen with the discrete Bayes filter algorithm. In fact, the Grid localization algorithms takes as input the discrete probability value $\{p_{t-1,k}\}$, along with the most recent measurement and control, and obviously the map. For each grid cell, at first it calculates an initial belief, and then this is updated through the information retrieved from sensors. In the most basic version of grid localization, the partitioning of the space of all poses is time invariant and each grid cell is of the same size. Also the choice of the dimensions of the grid cell is very important: smaller cells will leads to more accurate results, but also will increase the computational complexity of the operation. A common granularity used for indoor environment is 15 centimeters for x- and y-dimensions.

4.8 Graph-based

Graph-based SLAM is one of the last state-of-the-art technique used nowadays in SLAM applications. Its goal is the creation of a graph made up of robot's poses, represented by nodes, and constraints that connect the poses of the robot while moving. In order to achieve this result, a least square approach to SLAM is necessary.

4.8.1 Least Square

The *Least Square approach* is an approach for computing the solution of an over-determined system, minimizing the sum of the squared errors in the equations. Given a system described by a set of n observation functions $f_i(x)_i = 1 : n$, and considering x as the state vector, z_i as the measurement of the state x and $\hat{z}_i = f_i(x)$ as the function which maps x to a predicted measurement $\hat{z}_i$, given n noisy measurements $z_{1:n}$ about the state x, the goal is the estimation of the state x which best explains the measurements $z_{1:n}$.

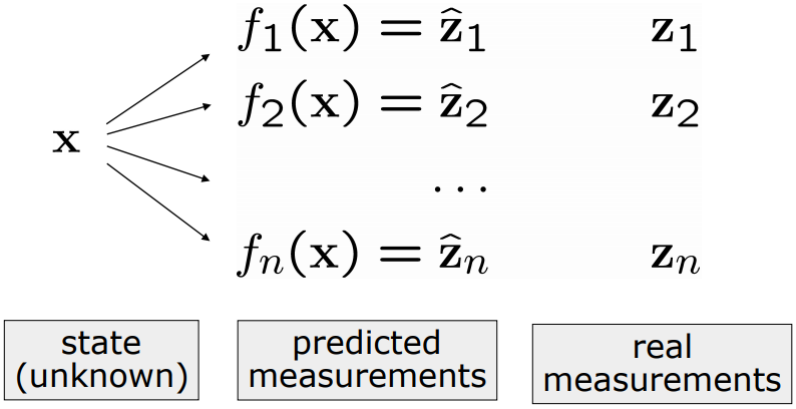


Figure 4.4: Least Square graphical representation [20].

These measurements are affected by an error e_i, which is the difference between the predicted and actual measurement $e_i(x) = z_i - f_i(x)$, and is generally considered with a zero mean and normally distributed. Remembering that Gaussian error can be represented by the information matrix Ω_i, the squared error of a measurement depends only on the state, and is a scalar that can be represented as:

$$e_i(x) = e_i(x)^T \Omega_i e_i(x) \tag{4.10}$$

And so, the goal of the least square approach is to find the state x^* which minimizes the error, that means:

$$x^* = argmin \sum e_i(x)^T \Omega_i e_i(x) \tag{4.11}$$

This problem is generally very complex and does not present closed form solution, so some assumptions are necessary:

- A good initial guess is available;
- The error functions are smooth in the neighborhood of the minima.

Now, a solution can be found through iterative local linearizations, by following the *Gauss-Newton solution*. Here, passages of the Gauss-Newton solution are listed, and later they are further exposed:

- Linearize the error terms around the current solution/initial guess. This will lead to the error equation

$$e_i(x + \Delta x) \simeq e_i(x) + J_i(x)\Delta x \tag{4.12}$$

- Compute the first derivative of the squared error function and find the terms for the linear system;

$$b^T = \sum_i e_i^T \Omega_i J_i \tag{4.13}$$

$$H = \sum_i J_i^T \Omega_i J_i \tag{4.14}$$

- Solve the linear system $\Delta x^* = -H^{-1}b$
- Update the state $x \leftarrow x + \Delta x^*$
- Iterate

Starting from the first point, the error functions are approximated around an initial guess x via Taylor expansion:

$$e_i(x + \Delta x) \simeq e_i(x) + J_i(x)\Delta x \tag{4.15}$$

Where J_i is the Jacobian and Δx is the increment. Now, replacing the Taylor expansion in the squared error terms it is possible to fix the initial estimate x, obtaining:

$$e_i(x + \Delta x) = e_i^T(x + \Delta x)\Omega_i e_i(x + \Delta x) \tag{4.16}$$

From which:

$$e_i(x + \Delta x) = c_i + 2b_i^T \Delta x + \Delta x^T H_i \Delta x \tag{4.17}$$

with

$$c_i = e_i^T \Omega_i e_i \tag{4.18}$$
$$b_i^T = e_i^T \Omega_i J_i \tag{4.19}$$
$$H_i = J_i^T \Omega_i J_i \tag{4.20}$$

This equation represent the squared error, with little modification it is possible to obtain the equation of the global error:

$$F(x + \Delta x) = c + 2b^T \Delta x + \Delta x^T H \Delta x \tag{4.21}$$

with

$$c = \sum_i c_i \tag{4.22}$$
$$b^T = \sum_i e_i^T \Omega_i J_i \tag{4.23}$$
$$H = \sum_i J_i^T \Omega_i J_i \tag{4.24}$$

This equation is the expression of the objective function, under a linear approximation of the sensor model around a neighborhood of the initial estimate x. Fixing the initial estimate, the value of the function in the neighborhood can be approximated by a quadratic form with respect to the increments Δx. Thus, it is possible to minimize this quadratic form and get the increment Δx^* that applied to the current guess gives a better solution:

$$x^* = x + \Delta x^* \tag{4.25}$$

But how can this increment Δx^* be found? Simply by finding the first derivative of $F(x + \Delta x)$ and setting it to zero. This lead to the equation:

$$\Delta x^* = -H^{-1} b \tag{4.26}$$

4.8.2 Graph-Bases SLAM

As said before, the main goal of Graph-Based SLAM is the creation of a graph, made up of nodes and edges. The building of the graph is commonly called front-end, and is heavily sensors dependent, while the second part called back-end relies on an abstract representation of the data, and has the goal of optimize the pose-graph to reduce the error.

The graph is build finding the node configuration that minimize the error introduced by the constraints and, it is easy to understand, this minimization is done

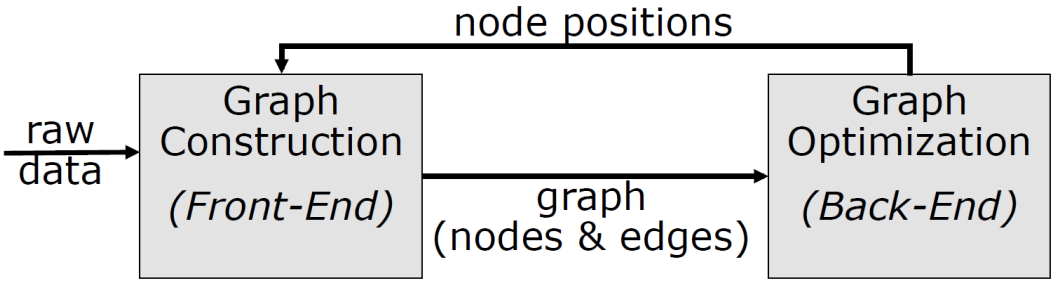


Figure 4.5: General Graph Based representation [21].

using the least square approach, in which the state vector x is represented by the nodes, meaning the poses assumed by the robot. Each x_i is a 2D or 3D pose, and a constraint/edge exists between x_i and x_j if the robot moves from one pose to the other. In this, the edge correspond to the odometry, so it is affected by noise, and its uncertainty is encoded by the information matrix Ω_{ij}, which is bigger the more edge matters in the optimization. Since nodes have different orientations, a transform matrix is useful to understand how one node is seen from the other. This transformation can be expressed using homogeneous coordinates:

$$(x_i^{-1} x_j) \tag{4.27}$$

Introducing the error on the single constraint $e_{ij}(x_i, x_j) = (z_i j^{-1}(x_i^{-1} x_j))$, with z_i, j the measurement of j with respect to i, it is also possible to represent the objective of graph-based SLAM of finding the node configuration that minimize the error:

$$x^* = argmin \sum_{ij} e_{ij}^T \Omega_{ij} e_{ij} \tag{4.28}$$

It is now possible to adapt the least square approach and the Gauss-Newton solution to achieve this goal. The error function around an initial guess x can be approximated via Taylor expansion as

$$e_{ij}(x + \Delta x) \simeq e_{ij}(x) + Jij\Delta x \tag{4.29}$$

with the jacobian J, equal to $J_{ij} = \frac{\delta e_{ij}(x)}{\delta x}$. It is worth noting that $e_{ij}(x)$ does not depend on all state variables, but only on x_i and x_j. This means that the jacobian will be non-zero only in the columns corresponding to x_i and x_j. These two columns can be denoted respectively as $A_{ij} = \frac{\delta e(x_i, x_j)}{\delta x_i}$ and $B_{ij} = \frac{\delta e(x_i, x_j)}{\delta x_j}$. This introduce a problem called *sparcity*, which is inherited by the structure of b and H. So, in this case, terms b and H can be represented as:

$$b = \sum_{ij} b_{ij} = \sum_{ij} J_{ij}^T \Omega_{ij} e_{ij} \tag{4.30}$$

$$H = \sum_{ij} H_{ij} = \sum_{ij} J_{ij}^T \Omega_{ij} J_{ij} \tag{4.31}$$

Which means b will be non-zero only at the indices corresponding to x_i and x_j, while H will be non-zero only in the blocks relating i, j. The resulting system in sparse, so it can be computed by summing up the contribution of each edge. For doing so, many solvers can be used, and this allow to efficiently solve the problem. In fact, once the error e_{ij} is computed, the coefficient can be retrieved as:

$$b_i^T += e_{ij}^T \Omega_{ij} A_{ij} \qquad b_j^T += e_{ij}^T \Omega_{ij} B_{ij} \tag{4.32}$$

$$H^{ii} += A_{ij}^T \Omega_{ij} A_{ij} \qquad H^{ij} += A_{ij}^T \Omega_{ij} B_{ij} \tag{4.33}$$

$$H^{ji} += B_{ij}^T \Omega_{ij} A_{ij} \qquad H^{jj} += B_{ij}^T \Omega_{ij} B_{ij} \tag{4.34}$$

And, as before, the increment Δx^* is obtained as $\Delta x^* = -H^{-1} b$.

4.8.3 Loop closure

Loop closure is a relatively recent, appearance-based localization and mapping technique used for improving both the localization of the robot and the construction of the 3D map. Loop closing is the act of correctly asserting that a vehicle has returned to a previously visited location [22], and update belief accordingly. This is achieved by recognizing and memorizing a certain number of features and locations in the environment, which will be unique to the specific pose of the robot, gather them into a set of descriptors, and comparing theme each other to determine similarity between local scenes. These sets of descriptors are often replaced by *bags of words*, a visual model that treat image features as a sparce vector of words. Loop closure integrates well with graph-based SLAM and has the capacity to greatly improve its results. In fact, When a loop closure hypothesis is accepted, a new constraint is added to the map's graph, then the graph optimizer minimize the errors in the map.

One of the most famous algorithm of this type is RTAB-Map (*Real-Time Appearance-Based Mapping*), a Stereo and Lidar Graph-Based SLAM approach based on an incremental appearance-based loop closure detector, which makes use of a bag of words approach and a memory management approach to limit the number of locations used for loop closure detection and graph optimization, so that real-time constraints on large-scale environnements are always respected [23].

Chapter 5

Software implementation

In this chapter, the implementation of the software used for the project is presented. It is divided in four parts, where the first two expose how both the deep learning room recognition convolutional neural network model and the simultaneous localization and mapping algorithm are obtained, while the third explains how these two have been integrated each other and with hardware components. Finally, the fourth part deals with how the algorithm has been further modified and refined in order to obtain the final results.

5.1 Room recognition

For the purpose of this thesis project, the robot has to roam in an unknown indoor environment and classify each different room it encounter. This is a clear example of classification problem and, as stated at Chapter 2.4.3, one of the most powerful algorithm used to solve such problems is the convolutional neural network. In the following sections, all the passages used in order to achieve the final room recognition algorithm will be presented:

- **initialization phase**, in which a CNN model is implemented and all the information and data useful for training are gathered;
- **training phase**, where the model is trained and tested;
- **Application**, where the model is exported and deployed on the robot.

For the implementation of the model, written in Python, many libraries are used. Here the main ones are listed:

- **TensorFlow**: it is a free and open-source software library use particularly for machine learning;

- **Keras**: an API of TensorFlow, is an open-source neural network library which provide a series of function for preprocessing of images and for implementation and training of CNN models;
- **scikit-learn**: an other very common free machine learning library [24];
- **GradCam**: an algorithm used to visualize on dataset's images the features extracted by the model during training.

5.1.1 TensorFlow and Keras

Working in a Python environment, TensorFlow is of fundamental importance in the creation of a Machine Learning model. TensorFlow is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML powered applications [25].

In this project, it is used alongside an intuitive high-level API, free and open-source, called Keras, which provide a series of helpful functions used for implement, train and test the model. Here, the fundamental ones are listed:

- `ImageDataGenerator`: create a data generator used in the following passages to import the image dataset. In it a series of parameters for data preprocessing can be specified, like color rescaling and image flipping;
- `flow_from_directory`: uses the precedently implemented data generator to import images from a specified folder. If images are divided in folders corresponding to each class, the label is automatically assigned to every image. Also in this case a series of parameters can be assigned in order to resize the imported images and divide them in batches of a specified size;
- `layers`: an API that use several functions to create different type of layer, the building blocks of neural networks. Some of these are `Dense`, which create a densely-connected NN layer, `Activation` for the application of activation functions; different type of convolutional layers like `Conv2D`, different type of pooling layers like `MaxPooling2D` and `AveragePooling2D` and many more;
- `Model`: a class that groups different layers in a neural network model. Once the model is created, it can be visualized with `model.summary()`, configured with losses and metrics with `model.compile()`, trained with `model.fit()`, or used to do predictions with `model.predict()`.

5.1.2 Data preprocessing

Before implementing the model, images of the dataset are preprocessed. *Data Preprocessing* is an important passage in machine learning: it consists in optimizing

data that will be used for training and validation of the model, allowing to reduce the computational complexity and increasing the accuracy of the algorithm. First of all, the dataset consisting of a certain number of images (and as many labels) is built and divided in training set and validation set, each of which is further divided into the same number of classes. For each class the ratio of images between train and validation sets is generally 3 or 4 to 1. This division is fundamental in which the model need a set to train and "learn" how to classify each image, and an other to validate such model, in a way presented at the end of Chapter 2.4.3. Since images of the dataset consist in RGB coefficient in the 0-225 range, which is too high for the model to process, with the *normalization*, they are rescaled by a 1/255 factor using `ImageDataGenerator`, to take them in the 0.0-1.0 range. Then, with `flow_from_directory`, they are divided in batches of 64 samples each, resized to a dimension of 224x224 pixels and, for the training set, they are shuffled in order to obtain better results during training. How explained before, the same function takes care also of the labelling division of the images in classes.

5.1.3 Checking model results

During the training and validation of the model, by default TensorFlow and Keras provide values of loss and accuracy for both training and validation. Often, these information are not sufficient to really figure out the behavior of the model, and further graphical visualizations are required to better understand if it is working as expected. An example is *Matplotlib*, a library for creating static, animated and interactive visualizations in Python. In addition to lots of other implementations, it can be very useful in the creation of graph depicting accuracy and loss curves, which is a more clear and immediate way to visualize the behavior of such metrics.

An other very useful tool is represented by *scikit-learn*. It provide two functions called `confusion_matrix` and `classification_report`. The former, as the name says, take as inputs the ground truth labels of a set of images and the labels predicted by the model on the same set, with which construct a confusion matrix. It is a square matrix with a number of row and columns equal to the number of classes. Every row correspond to the predicted classes, while every column correspond to the true classes. This means the diagonal will represent the number of points for which the predicted label is equal to the true label, while off diagonal elements are those that are mislabeled by the classifier. This visualization can be very useful for binary classification problems, or multiclass problems where the number of class is restrained.

The `classification_report` function takes as inputs the same parameters of confusion matrix, plus the names of each class in the set. Then, it build a text report showing the main classification metrics for each class and for the whole set. This visualization is more complete, providing more data than confusion matrix, and is a better choice for problems with lots of classes.

Figure 5.1: Generic Matplotlib metrics behavior visualization.

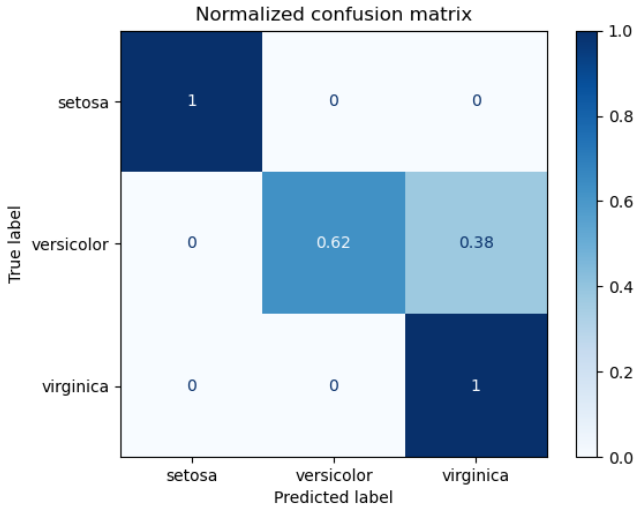


Figure 5.2: Generic scikit-learn Confusion Matrix [26].

Furthermore, there are also techniques able to making the CNN-based model more transparent by directly visualizing images of the dataset and providing a series of information on them, helping the user to understand how the classificator behaves with each one. One of these is Grad-CAM (*Gradient-weighted Class Activation Mapping*), a technique able to retrieve and highlight, with a coarse localization map, the more "important" regions of input for prediction in every dataset's image [27]. In other worlds, it visualizes, for each image, the regions where the model focuses more in order to provide a prediction.

5.1.4 Model implementation

Once the dataset is loaded and ready, it is possible to proceed to the implementation of the model. It is worth mentioning that, like for any other machine learning work, several attempts have been made before obtaining the final model. Trial and error is fundamental for parameters and hyperparameters optimization, which directly impact model metrics. For the sake of concisivness and practicality, in the next paragraphs, only the salient implementation attempted before reaching the final room classification model are presented. Anyway, the loss function employed has never been changed, is always a `categorical crossentropy` function, very adapt for multiclass classification problems as this. For the same reason, the internal layers' activation function will be always a ReLU function, while the last, output layer will always have a softmax function.

In a very first stage of the project, a really elemental convolutional neural network model, created from scratch, was tried. It was a sequential model constituted of a dozen layers, alternating between `Conv2D` and `MaxPooling2D` in the first layers, terminating with some `Dense` layers, and running on 10 epochs. Obviously the performance of this model was very limited under many aspects: first of all, both training and validation accuracy was very low, hovering around values of 30%, yet there was an overfit problem. Increasing the number of epochs only made the overfitting worst, but accuracy values did not improve. The lack of accuracy was clearly caused by the simplicity of the model, respect to the complexity of the scenes depicted in the dataset's images. An other problem, and reason of both low accuracy and overfitting, was the structure of the dataset. In this first implementation, it was constituted of images retrieved from some classes of the MIT's Indoor Scene Recognition CVPR 09 dataset [28]. The dataset used presented 11 classes, each containing at most a few hundred of images. Practically, there were too few images for too many classes to expect better results. This factor also influenced the result on the loss values, which were very low.

In that moment, the primary focus was to improve accuracy and loss results: it is not worth spending time and resources in trying to reduce overfitting when also the training accuracy is so low. Hence, in a second time implementation, it was

decided to start from a pre-trained existing model, and the choice fell on ResNet50 [29], a convolutional neural network 50 layers deep. Some final layers were added to it, in order to adapt it to the case of study of this project, and the layers of the whole obtained model were set as trainable.
Furthermore, the dataset was downsized to 7 classes, simplifying the model classification. Running on 10 epochs, this model gave remarkable better results compared to the precedent model, but yet not good enough to be considered acceptable. Training accuracy was very high, with values over 90%, but on validation it stagnates under 60%, in an obvious overfitting manner. The same behavior could be visible on losses, with values on training very low, but not as low on validation.
It was clear that the next step to go was to try to reduce the overfiting, hoping it would improve accuracy on validation set as well.

After a series of unsuccessful attempts to reduce the overfitting on the model, involving the use of more advanced hyperparameters and data augmentation techniques, it was decided to dismiss that model based on RedNet50 and try using a total different model. To that end, MobileNet was chosen, a pre-trained CNN model that is 53 layers deep [30]. Also in this case, some `GlobalAveragePooling2d`, `BatchNormalization` and `Dense` layers where added at the bottom to adapt the model. In addition, some hyperparameter settings used for reduce overfitting with the precedent model were maintained, and while on that model they had not given great results, in this case the situation has improved. Such hyperparameters that enhanced the model are the learning-rate of the `Adam` optimizer, which give better results if reduced (but not too much), and the addition of some `Dropout` layers after `Dense` layers.
After all this implementation, the new model performed incredibly well: in fact, not only accuracy values increased over 83%, but also losses were greatly reduced. In any case, overfitting was still too influential, and there was room for improvements, yet.

In an attempt to reduce overfitting and further increase accuracy, a new dataset were created, integrating the CVPR 09 with much more samples, obtaining a final dataset based on 6 classes, *Bathroom*, *Bedroom*, *Closet*, *Dining room*, *Living room*, and *Kitchen*, for a total of over 7000 images, which was a great improvement compared to the 2700 circa used previously. Running the same model as before with this new collection of images, accuracy further increased, peaking values of 87%. However, here arised a new problem: the whole system was very unstable, and if sometimes final accuracy results were so high, other times they were just as low, reaching values below 65%. Besides, also when these values were high, on losses was evident the presence of a heavy overfitting problem that caused validation loss to be much more high compared to training loss.

Finally, a last successful attempt was made to solve the problems reported above. A small modification to the model structure was sufficient to improve its results, and led to the final model used for this project. Practically, more `Dense`

followed by `Dropout` layers were added, and on `Dense` layers *Kernel regularizer* was implemented. This hyperparameters works trying to reduce the weights, that means the Kernel matrix (for reference see section 2.4.3).
Now, this model reaches accuracy results on training set over 85%, which is slightly lower than before, but is the price to pay for having a much more robust system, and much better loss results which are fairly close between training and validation, as can be seen from Figure 5.3. The overfitting problem is been greatly reduced, although not completely solved. Anyway, these results can be considered pretty good, given basically two factors: first, the scenes shown in the images do not depict single objects but whole rooms, and are quite complex, and in general difficult to be analyzed by a neural network and, secondly, for the purpose of this project a number of instances potentially infinite can be run in every room, this makes the single prediction lose value in favor of a more complete general picture.

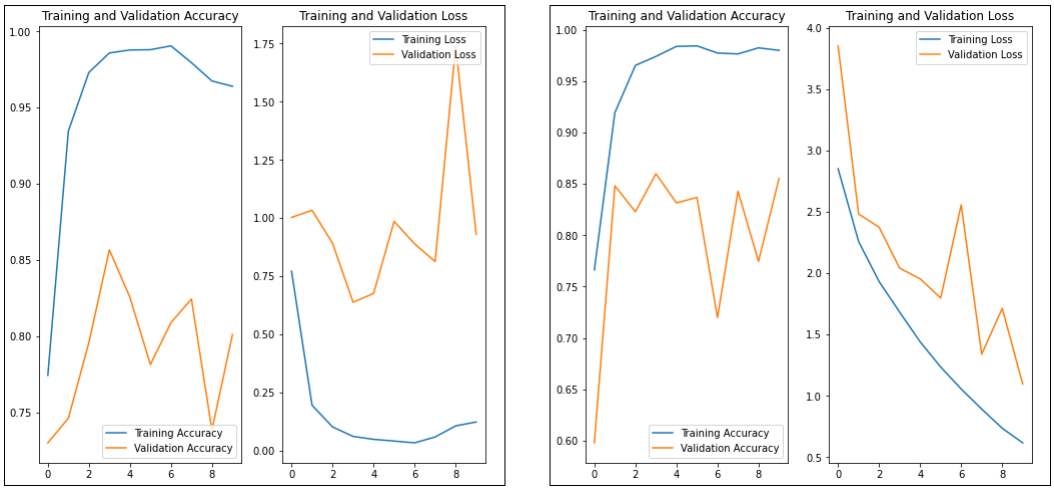


Figure 5.3: Visualization of results of the actual classification model (on the right) compared to the previous one (on the left).

Below it is possible to observe the confusion matrix and the classification report obtained for the final model.

Some of the most peculiar results obtained with Grad-CAM on the room recognition model are also presented.

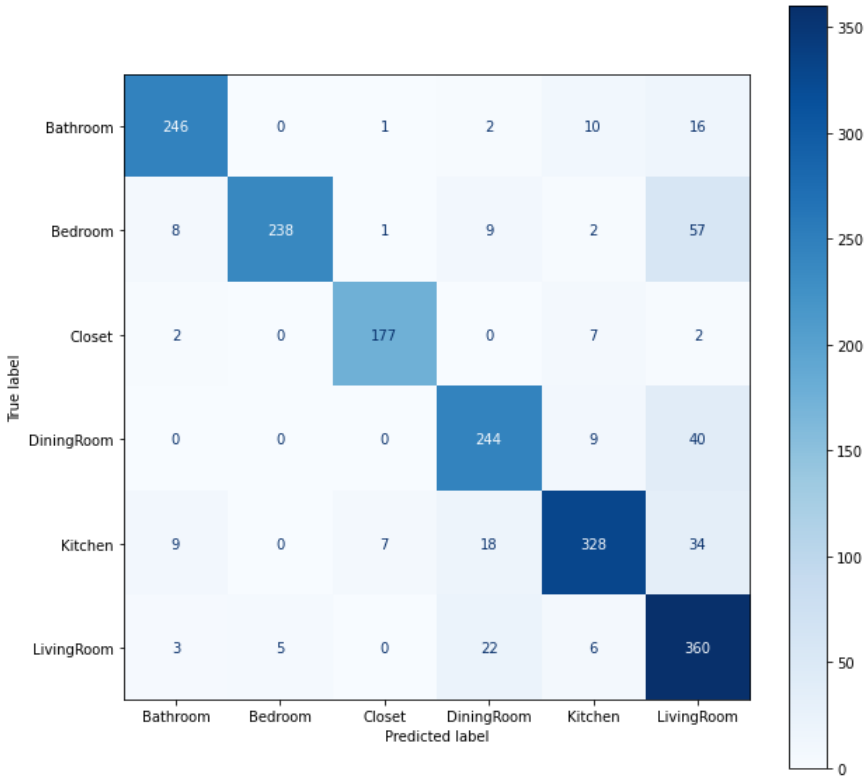


Figure 5.4: Confusion Matrix of the final room recognition classification model.

	precision	recall	f1-score	support
Bathroom	0.92	0.89	0.91	275
Bedroom	0.98	0.76	0.85	315
Closet	0.95	0.94	0.95	188
DiningRoom	0.83	0.83	0.83	293
Kitchen	0.91	0.83	0.87	396
LivingRoom	0.71	0.91	0.80	396
accuracy			0.86	1863
macro avg	0.88	0.86	0.87	1863
weighted avg	0.87	0.86	0.86	1863

Figure 5.5: Classification Report of the final room recognition classification model.

5.2 SLAM algorithm

How presented in Chapter 4, lots of different SLAM algorithms and techniques exist, and it is not always simple to figure out which of these might best fit the problem in question. For this thesis project, it is needed a fast but lightweight program, able to run on the NVIDIA Jetson alongside the room recognition CNN algorithm and to operate in real-time. Furthermore, it should be integrable with

Figure 5.6: Some results of the analysis conducted with Grad-CAM.

ROS and its environment, easily implementable with RealSense cameras, and also should be able, at the end of the process, to generate a 2D map of the environment on which the room recognition predicted annotation could be written.

5.2.1 Kimera

One of the first implementations attempted for this project, exploited an advanced and at the state-of-art software developed by MIT, called *Kimera*. Kimera is a C++ library for real-time metric-semantic simultaneous localization and mapping, which uses camera images and inertial data to build a semantically annote 3D mesh of the environment. Kimera is modular, ROS-enabled, and runs on a CPU [31].

This modularity derive from the inclusion of four key module:

- **Kimera-VIO**: a VIO module for fast and accurate IMU-rate state estimation;
- **Kimera-RPGO**: a robust pose graph optimization method that adds a robustness layer which avoid SLAM failures;
- **Kimera-Mesher**: a module that computes a fast per-frame and multi-frame regularized 3D mesh to support obstacle avoidance;
- **Kimera-Semantics**: a module that builds a slower-but-more-accurate global 3D mesh using a volumetric approach.

While this implementation can seem optimal for the studied case, a series of problems occurred during its implementation: first of all, although Kimera is really lightweight compared to all the services it offer, it is not light enough to be efficiently run on Jetson AGX Xavier, and its employment alongside the classification algorithm might be very problematic. Secondly, while it allows to obtain a 3D global mesh of the environment, there are no currently ways to retrieve a 2D map.

5.2.2 RTAB-Map

Since Kimera appeared not quite appropriate for the case of study, on a second implementation it was decided to use RTAB-Map, for a series of reasons reported below:

- It is relatively lightweight in terms of computational effort, that makes it adapt for embedded system like the one in use;
- It is specifically designed for real-time operations, thanks to its memory management approach which limits the number of locations used for loop closure detection and graph optimization;
- A ROS wrapper is available, making it well integrable with the ROS environment and easily implementable with `rviz`;
- It is well integrable with Intel RealSense cameras, in particular with the active IR stereo system of the D435i;
- In addition to its ability to generate a 3D point clouds map of the environment, which can be useful in an implementation phase to monitor the system behavior, it provide also the construction of a 2D occupancy grid map, which is the best desired output for this project.

The ROS wrapper for Intel RealSense Devices [32] provides several packages suitable for using RealSense cameras with ROS. Among others, it provide a `roslaunch` called `opensource_tracking`, which allow to use RTAB-Map for conducting a SLAM operation with the RealSense D435i as the only sensor. For doing this, it exploits, in addition to RTAB-Map and RealSense cameras initialization packages, other two powerful open source tools called `imu_filter_madgwick` [33] and `robot_localization` [34]. The former is used to filter and fuse raw data from IMU devices, the latter provides nonlinear state estimation through sensor fusion.

Even though this first implementation seems to work pretty well, sometimes it proves not very robust: how discussed at Section 3.3.2, the D435i IMU, on its own, is not always able to ensure a tacking stability good enough, even when integrated with some powerful tools like those mentioned earlier; hence the introduction of the Intel RealSense T265, which supplys a very precise VIO, is fundamental. With this addition, codes must be changed for adapting the two cameras to RTAB-map.

Intel already provides a ROS launcher file that initialize both the T265 and the D435i, called `rs_d400_and_t265.launch`. This launcher create some nodes, each of which publish several topics. Among these, the most interesting one are listed below:

- `/d400/realsense2_camera_manager`: a node related to D435i camera, which publishes:

  - `/d400/color/image_raw`: stream the image received by the color camera;
  - `/d400/depth/color/point`: stream a depth point cloud which respect the RGB code of the image received by the color camera;
  - `/d400/aligned_depth_to_color/image_raw`: stream a depth image. While conceptually it is different from `/d400/color/image_row`, which is a PointCloud type data instead of a depth image type data, if implemented alongside `/d400/color/image_row`, from a visualization point of view they can be almost equivalent;

- `/d265t/realsense2_camera_manager`: a node related to T265 camera, which publishes the topic `/t265/odom/sample`, directly related to the camera VIO, used by RTAB_Map for localization and tracking, instead of D435i's IMU of the precedent implementation. In addition, the use of T265's VIO makes unnecessary the integration with `imu_filter_madgwick` and `robot_localization`, which are then removed;

On the other side, the ROS wrapper for RTAB_map provides a launcher file simply named `rtabmap`, which launch the `/rtabmap/rtabmap` node, which publishes the following topics:

- `/rtabmap/grid_map`: takes depth data from sensors and generate an occupancy grid map of the environment;
- `/rtabmap/octomap_grid`: takes depth data from sensors and generate an occupancy grid map of the environment called *octomap*, because it is based on *Octree*, a tree data structure in which each internal node has exactly eight children. It is faster than a normal grid map, but not as precise;
- `/rtabmap/mapData`: takes data streamed by a depth point cloud and construct an RGB, 3D point cloud map of the environment.

A `rviz` configuration is created, in such a way that when it is launched alongside the SLAM algorithm the most relevant information can be immediately visualized, removing some useless parameters and adding others, enabling the main parameters and leaving disabled some less significant alternatives. Moreover, the map frame of the environment is selected as fixed frame, while among all the possible mobile frame it is chosen the frame integral with the T265's camera, `t265_link`, which is also the main reference for the localization and tracking of the system. All the other reference frames (such as `t400_link`, integral with D435i camera) are hidden in favor of a clearer visualization.

At this point, in order to implement a immediate and user-friendly program able to combine these two systems, a new ROS launcher file is created. It launches both `rs_d400_and_t265.launch` and `rtabmap`, but the latter needs some modification to fit the current problem:

- `/t265/odom/sample` is passed as odometric topic, so that RTAB-Map can use T265's VIO for localization and tracking, and the reference frame `t265_link` is passed as the main reference frame of the mobile system. As a consequence, the parameter `visual_odometry`, which introduce a visual odometry software implementation in the absence of a VIO hardware sensor, is set to be disabled;

- `/d400/aligned_depth_to_color/image_raw` is passed as the depth image topic, `/d400/color/image_row` is passed as the RGB image topic, and `/d400/color/camera_info` is passed as the `camera_info` topic;

- The `rgbd_sync` parameter is set to be used. It synchronize RGB, depth and `camera_info` messages into a single message, which is useful when, for example, rtabmap is subscribed also to a laser scan or odometry topic published at different rate than the image topics. As a consequence, `approx_rgbd_sync` is set to be disabled;

- Since, by default, RTAB-Map use a proprietary software for visualizations called *rtabmapviz*, it is set to not be utilized in favor of `rviz`. In addition, the `rviz` configuration created before is passed as default one when the algorithm is launched.

The launcher thus obtained, is able to initialize both Intel RealSense cameras and RTAB-Map, and immediately start visualizing the RGB image streamed, the depth point cloud and the occupancy grid map of the scene within the camera field of view, alongside other parameters which can be easily activated when needed (for reference, see Figure 5.7).

5.2.3 Gmapping

As interesting and innovative as RTAB-Map may be, it presents a significant problem. The 3D map obtained by the algorithm is an impressive results, considering it is built only with a commercial depth camera. The same can be said for the occupancy grid map, but unfortunately the latter is still too dirty for the purpose of this project. Although until now it has been tried to avoid using devices other that the depth camera for the construction of the map, in order to obtain a better representation of the environment, the use of the lidar module integrated in the TurtleBot3 Waffle proved necessary.

Since the Lidar alone is actually able to provide a pretty much accurate occupancy grid map without the use of other devices or advanced software like RTAB-Map, it is decided to lean on simpler but also more lightweight mapping software. A series of these are available and compatible with the Waffle lidar, like for instance *Gmapping*, *Cartographer*, *Hector*, *Karto*, and so on. For this project it was decided to use Gmapping, but they all are quite equivalent and interchangeable for this application. Gmapping is a mapping software based on a particular particle filter,

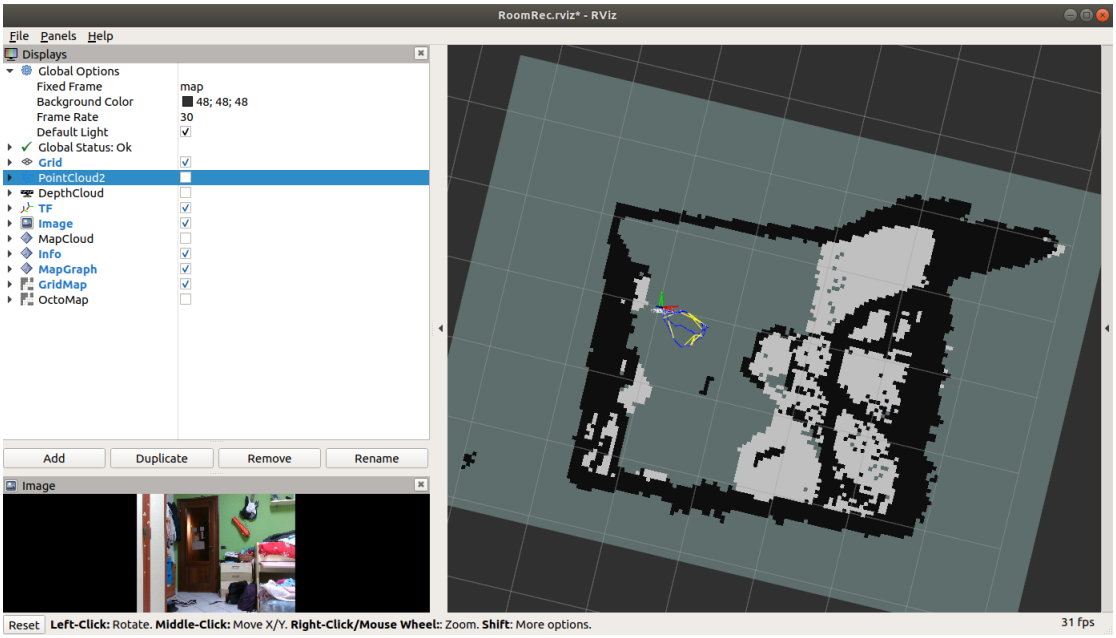


Figure 5.7: Visualization of the working RTAB-Map SLAM algorithm in `rviz`. As default, only the map grid (**Grid**), the depth point cloud (**PointCloud2**), the main reference frames (**tf**), the RGB image (**Image**) and the occupancy grid map (**GridMap**) are activated, but other parameters can be enabled at any time. These are the RGB depth image (**DepthCloud**, see above for the difference with PointCloud2), the global 3D environment's point map constructor (**MapCloud**) and the occupancy grid octomap (**OctoMap**).

called Rao-Blackwellized particle filter, in which each particle carries an individual map of the environment [35]. ROS already provide a node able to initialize and run a SLAM algorithm, which at its base has Gmapping as mapping program, called `slam_gmapping`. This node takes as input the scan message from the lidar, and provide as output the occupancy grid map. As said previously, Gmapping is definitely simpler than RTAB-Map, it cannot create a 3D map of the environment, and is not based on a SLAM technique as sophisticated, but for the purpose of this project it is not a big deal.

If the mapping part of this process seems really linear, this implementation does not solve the other great problem of a SLAM algorithm, that of localization. In fact, in order to localize the robot during the mapping, the Gmapping node requires also the three reference frames of lidar, robot and map (that is the one of the physical world or environment in which the robot has to navigate), and the transformations between them. Additionally, one of these reference frames has to track the robot during the navigation. Gmapping is virtually able to perform this tracking using the information coming from the lidar scan, but like the D435i, this tracking system is not very precise and subject to gross errors. But similarly to the

RTAB-Map case, this problem can be easily bypassed using the VIO information coming from the Realsense T265. Once the camera is mounted on the robot, both their reference frames can be considered integral each other, and that of the lidar as well. Once this block is defined, it is sufficient to define its initial position with reference to the map, and all the subsequent poses assumed by the robot during the navigation will be automatically computed by the SLAM algorithm.

As before, a new ROS launcher file is created. This file, simply called `robot_slam.launch`, alongside `rs_d400_and_t265.launch`, initialize four other nodes:

- `slam_gmapping`, presented previously, able to start the SLAM algorithm based on Gmapping. A couple of arguments are passed to the node in order to define the `/scan` topic as the lidar scan argument and the `t265_pose_frame` topic as the base frame of the robot. In this case, the `t265_pose_frame` topic plays the same role as `t265_link` of the previous implementation, furthermore passing it as the base frame automatically solve the problem of setting the camera reference frame integral with that of the robot;

- Two `static_transform_publisher` nodes, which define a static transformation between two reference frames. The first is used to set the map frame equal to that of the initial position of the robot, and the second set the lidar integral with the robot;

- As before, the last node run a custom `rviz` configuration, reported in Figure 5.8.

The Intel Realsense D435i, although is not necessary anymore for mapping, is maintained for its RGB camera feature, very helpful for the classification part.

5.3 Integration

The classification and the SLAM algorithms have to be integrated in a single one, able to guide an unmanned terrestrial robot in a domestic, unknown indoor environment, mapping it and making a certain number of prediction for each room. However, while the SLAM algorithm is already in ROS language, so it can easily be integrated with a robot, the room recognition classification model currently available is unable, by itself, to perform all a series of fundamental operations, like communicate with the ROS system, retrieve the images to classify, provide the classification result and so on. In order to achieve all these tasks, it is necessary to integrate first the previously obtained model with the ROS environment, and then with the localization and mapping algorithm.

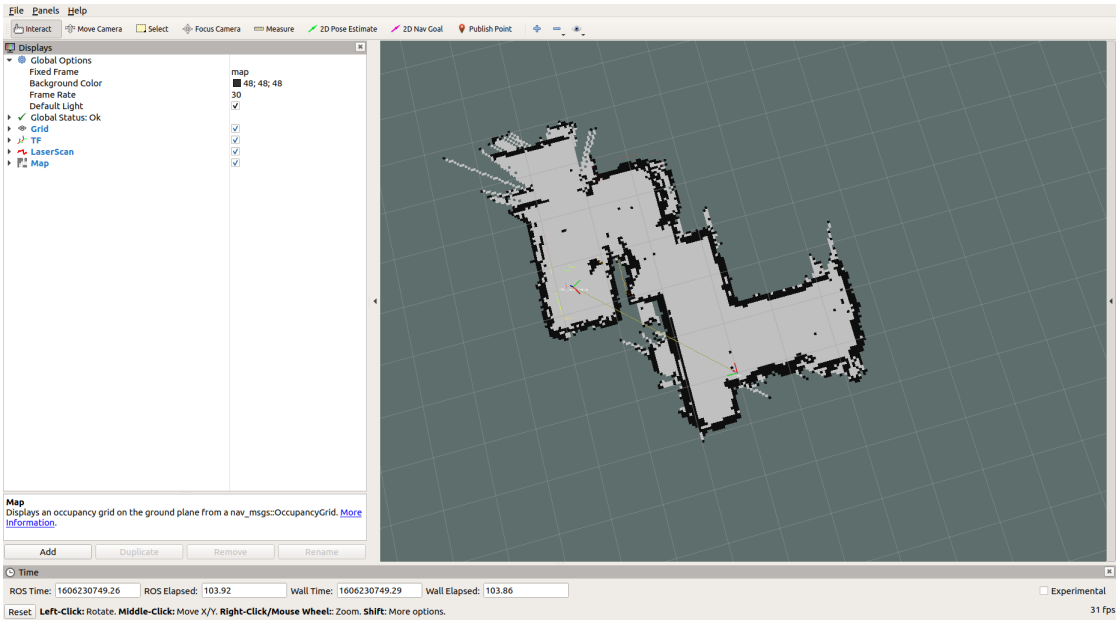


Figure 5.8: Visualization of working SLAM algorithm with Gmapping in `rviz`. Alongside with the grid (**Grid**), reference frames (**tf**) and occupancy grid map (**Map**), there is also the visualization of scan data received from the lidar (**LaserScan**).

5.3.1 Acquire and predict

To recognize the rooms encountered by the robot, this has to acquire some images of the environment as it advances, and provide them to the model. These images must have a resolution good enough and must be in a RGB format to be properly used by the model, and the RGB camera provided with the Intel RealSense D435i is perfect for this purpose.

For the acquisition of frames streamed by the color camera, Intel provide an open source computer vision library called *OpenCV* (Open Source Computer Vision Library), written in `C++` but well integrated in Python [36].

Since every vision sensor device connected to a machine running Linux is associated to a unique ID and number, OpenCV provides a function called `VideoCapture()` which, specifying the number of the desired camera, is able to directly access to such device and stream, with a function called `imshow()`, the images acquired by it in a window onscreen. In such a way all the frames captured by the camera are received, with a frequency very near to the frame rate of camera. For the purpose of this project, the model has to predict only one image for each unit of time, so it is possible to set a timer and, after every quantity of time, save a frame for the prediction. Ideally, it would be easy to pass such frame directly to the model, and use the TensorFlow function `predict()` to obtain a prediction. Actually, to be

processed by the model, that frame has to be preprocessed, exactly how it is done for the image dataset before the training. So it is much more easy to save that frame with the OpenCV function `imwrite`, specifying the name and the directory location of the image, re-load it with the TensorFlow function `load_image()` which allow also to resize the picture during its importing, add it to an array, normalize by dividing by 255, and then use it for the prediction.

Now, the algorithm described above is very simple and can work if executed by itself, but it cannot, in any way, communicate with other ROS components, because in general it is not integrated in a ROS system. Initially, this may not seem like a big deal, but what would happen if a ROS process which use the same D435i camera (like, for example, the SLAM algorithm) is run concurrently? In that case, the camera is set as a mutually exclusive resource for the ROS system, which means only components of the ROS system can *see* it. That means the acquisition and classification algorithm, as it is now, cannot use that camera anymore. Or rather, it cannot until it is integrated in the ROS system.

In order to achieve this goal, since, how is described before at Section 5.2.2, the ROS launcher `rs_d400_and_t265.launch` publish a series of topics providing different resources of the D435i camera, like the image received by the color camera, it is possible to subscribe to the relative topic, called `/d400/color/image_raw`, and retrieve the image frames from it. But here arise a new problem: these images are in ROS image message format, so OpenCV can't process them directly, unless *CvBridge* is used. CvBridge is a software able to convert ROS images into OpenCV images and vice versa.

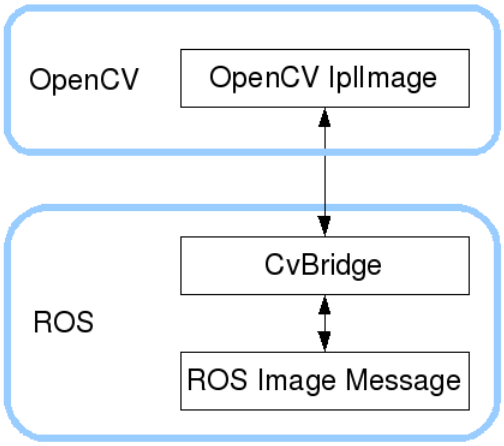


Figure 5.9: CvBridge structure visualization [37].

Hence, all is needed to do, is to create a new node, subscribe it to the color camera topic, and then use the CvBridge function `imgmsg_to_cv2` to convert the

camera frames into OpenCV images. At this point the process is very similar to before: after each unit of time, an image is saved, re-loaded, preprocessed and then is used to make a prediction. As now, results of predictions cannot be exported, they can at most be printed on the system terminal.

5.3.2 Set the Marker

Before dealing with importing that data back into the ROS system, an other source of information is necessary. The prediction result has indeed to be placed on the environment map, at the location assumed by the robot when the image has been taken, so the algorithm must also retrieve the robot odometry information. For this purpose, the same node created before, is subscribed to another topic, the T265's `/t265/odom/sample`. In such a way, it is possible to retrieve the pose of the robot, so that, after each unit of time, in addition to save the image for the prediction, simultaneously it also saves the position of the robot in that moment.

Now that the prediction result and the position where to place it are retrieved, it is necessary to find the answer to two fundamental questions: *"How to import that data in the ROS system?"* and *"How to visualize it?"*.

Starting from the second question, `rviz` provide a very useful tool called *marker*, which allow to add some annotation of various nature on the visualization of the map. These markers are displayed in a 3D spot in the world, and can be geometric shapes, like cubes, spheres and pyramids, or can be convenient text annotations, that always appears oriented correctly to the view, which are exactly what is needed for this project. So, an array of text markers is created (it is more convenient to use an array of markes rather than multiple markers because a single marker entity is always less expensive to render), and the shape, dimensions and color is selected. After each unit of time, a new marker is added to this array, and its location in the map is set to be each time equal to the actual position of the robot in that moment. Obviously, the text of each marker is set to correspond to the prediction result of the image taken at that moment, in that location.

Finally, to the question *"How to import that data in the ROS system?"*, the best answer is *"In the ROS way"*. This means creating a publisher topic, called `marker_topic`, which, after each unit of time, publishes the marker data generated at that moment; then `rviz` subscribes to that topic in order to visualize on the map all the annotations constituting the marker array.

Summing everything up, when the ROS `rs_d400_and_t265.launch` is launched, so that the Intel RealSense D435i can broadcast RGB image data and T265 odometry data, this algorithm create a node that publishes a marker topic and subscribes to the color camera and to the odometry topics, then uses the information thus received to create and publish a marker array to which, after each unit of time, is

added a new text marker, located in the same position assumed by the robot at that time, and displaying the prediction result of the image taken at that moment, in that location.

5.3.3 Deploy on machine

Moving to the robot, it is necessary to initialize all of its sensors and actuators. A new ROS launcher is created, named `start_robot.launch`. In turn, this launches three other ROS files:

- `turtlebot3_robot.launch`, a bring up package provided for TurtleBot3 applications. When a ROS master is running on a host computer (which is intended to be used for teleoperate the robot), this node is able to put in communication the ROS systems running on the two machines (the robot and the host pc). It initialize the lidar module mounted on the TurtleBot, and passing the scan argument it is automatically able to associate scan information to the corresponding topic. Moreover, it initialize the motors for the movement of the wheels. In such a way, when `turtlebot3_teleop_key.launch` will be launched on the host computer, it will be possible to actually control the robot;
- `rs_d400_and_t265.launch`, which, as said before, initialize the two Intel Realsense cameras.

Subsequently, in the previously presented `robot_slam.launch` is inserted the classification algorithm discussed in Subsection 5.3.1/5.3.2, so that with a single launcher it is possible to run the whole localization, mapping and classification algorithm.

The program thus obtained is robust, easy to execute, quite precise and relatively lightweight, when executed on a PC. But since it has to run on the robot, it is necessary to deploy the whole algorithm on the NVIDIA Jetson AGX Xavier, and here some problems arise. In fact, while the Jetson can run the two algorithms (the SLAM and the classification ones) individually, it is computationally not efficient enough to execute both together. After a more careful analysis, the problem is to be found in the room recognition algorithm, which requires a considerable amount of RAM memory.

TensorFlow provide a framework for on-device inference, designed for mobile and embedded devices, called *TensorFlow Lite*. With the *TensorFlow Lite Converter*, a tool avilable as a Python API, it allows to compress a TensorFlow model in a reduced and faster model, which does not affect accuracy, and optimize it in various ways.

The room recognition model used until now is then taken and converted in two different way:

- The first obtained model is simply a direct conversion from TensorFlow to TensorFlow Lite;
- The second one is obtained like the first one, but then it is optimized with a process of *quantization*, which convert the 32bit model in a 16bit. This operation reduces the precision of values and operations within a model, but also the size and the time required for inference.

The first converted model is reduced from 64MB of the original model to a bit more than 20MB, while the second is just 10MB. These two model are then tested with a test dataset and, while both are a lot more lightweight and less resources demanding, the quantized one proves to be more accurate to the other one, and gives results almost identical to the original room recognition model.

For this reason, the quantized model is chosen and replaced to the original model in the classification algorithm, and the whole program is retested on the NVIDIA Jetson AGX Xavier. In this case, the execution gives satisfying results, almost equal to those obtained on PC.

5.4 Final refining

Although this algorithm is able to perform quite well, a series of problems are still present:

- Even though the integration with lidar allows a clearer visualization of the map of the environment, this map is still affected by some errors and discrepancy;
- Publishing a marker for each prediction can be very problematic, since if just a minimal part of them reveals wrong, it can cause a very confusing interpretation of the effective final result of the classification;
- Anyway, all these markers normally visualized in `rviz`, cannot be saved on the final representation of the map;
- As lightweight as the algorithm is, sometimes it can still crash or block its execution;

While these problems arise from different causes, with a specific workaround it is possible to solve all of them in one go. This solution consists in dividing the whole algorithm, currently fully executed in real time, into two steps:

- In a first step, the robot navigates the environment. At this time only the strictly necessary information is gathered;
- In a second step, the previously acquired data undergoes a post processing operation to obtain final results.

5.4.1 First Step: Real Time Operations

In this first step, after `start_robot.launch` is run and the teleoperation service is initialized from the controller computer, `robot_slam.launch` is executed. It is the same implemented at Subsection 5.2.3, with the only difference it launches also the node `data_acquisition.py`. This node, at each time unit, acquires an image frame from the `/d400/color/image_raw` topic and save it in a specific directory. For each image it acquires also the position assumed by the robot at the time that image was acquired, and saves all these positions in a CSV file. When the mapping process of the environment is complete, running `save_and_print.launch` it is possible to save the occupancy grid map just created.

As it can be noted, in this process the only resource consuming operation is the SLAM algorithm. The acquisition of images and positions is situational and, in terms of computational efficiency, almost negligible. In this way during the real time operation, all the resources are available to the localization and mapping program, with no other execution running which can interrupt or interfere in any way with the creation of the map.

5.4.2 Second Step: Post Processing Operations

In a second step, when the robot has finished its journey in the environment and the SLAM algorithm is shut down, `Predict.launch` is executed. This launcher runs two different nodes:

- `X_Y_Pred_generator.py`, which imports images and positions data collected in the previous step, then use the room classification model to makes predictions about the images, and returns a vector containing, for each position of the robot, the estimate of the image taken in that position. This vector is then saved in a CSV file called `X_Y_Pred.csv`;

- `Mark_map.py`, which is indeed contained into another launcher called `Mark_map.launch`. In `Predict.launch`, the execution of this launcher and the subsequent node is delayed by ten seconds, so that `X_Y_Pred_generator.py` can finish its execution.

  This node retrieves the raw map constructed before and, in an operation of post processing, applies a series of computer vision technique and filters in order to enhance the quality of the visualization and eliminating any gross error. The map is then enlarged in scale, to allow a better and user-friendly visualization. The `X_Y_Pred.csv` is imported and a temporal mean is performed, so that for every n position/prediction samples, a new position/prediction sample is obtained, which is the mean of these n samples. Each prediction of these newly obtained samples is printed on the processed map, in the position associated with it.

The final result is an high resolution version of the map, which present a series of labels used to identify each room of the domestic environment.

Chapter 6

Results

In this chapter, the main results obtained during the development of this project are presented, and at times compared with the final results. Furthermore, major issues which still afflict the final implementation are exposed and a solution is proposed.

6.1 Mapping and data acquisition

Summarizing the subject of the last chapter, the final algorithm obtained runs completely on the NVIDIA Jetson AGX Xavier mounted on the TurtleBot3 Waffle, with the use of an external computer for teleoperation. Once these two machine are set in communication, and the `roscore` command run on the host computer, `start_robot.launch` is executed on the robot to initialize the bring up. Launching `turtlebot3_teleop_key.launch` on the host computer is now possible to control and drive the Waffle. At this point it is possible to start the slam operation launching `robot_slam.launch`. `data_acquisition.py`, associated to it, gathers frames in a directory called *acquisition*, and positions in a vector. When the node is stopped, it save such a vector in a CSV file called `robot_pose.csv`, similar to that exposed in Figure 6.1.

Before closing `robot_slam.launch`, launching `save_and_print.launch` it is possible to save the image of the map in a file called `map.pgm`, and its major information in `map.yaml`. An example of this file is reported in Figure 6.2: the most important information is contained in the first three lines. The voice *image* refers to the name of the original map image, *resolution* indicates the ratio between the dimensions of a pixel on the image and the real ones. In this case every pixels has a dimension of five centimeters in the real world. Finally, *origin* denote, in meters, the position of the origin of the image with respect to the origin of the map, defined during the mapping operation.

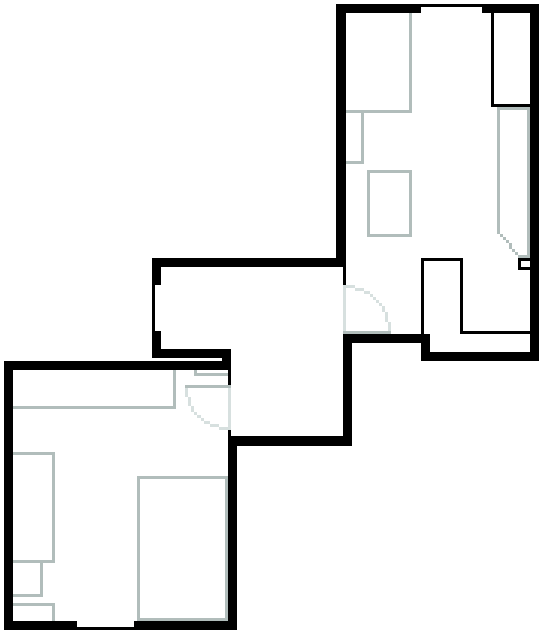
1.97E-01	7.56E-01
8.77E-01	5.25E-01
1.29E+00	4.47E-01
1.67E+00	5.22E-01
2.00E+00	7.20E-01
2.44E+00	9.39E-01
2.76E+00	1.02E+00
3.08E+00	1.04E+00
3.44E+00	1.06E+00
3.82E+00	1.08E+00
4.12E+00	1.17E+00
4.20E+00	1.48E+00
4.20E+00	1.52E+00
4.16E+00	1.87E+00
4.08E+00	2.23E+00
4.00E+00	2.52E+00
3.93E+00	2.62E+00
3.68E+00	2.81E+00
3.52E+00	2.86E+00
3.28E+00	2.90E+00

Figure 6.1: Sample of CSV file created by `data_acquisition.py`. The first column corresponds to x coordinates, the second to y coordinates.

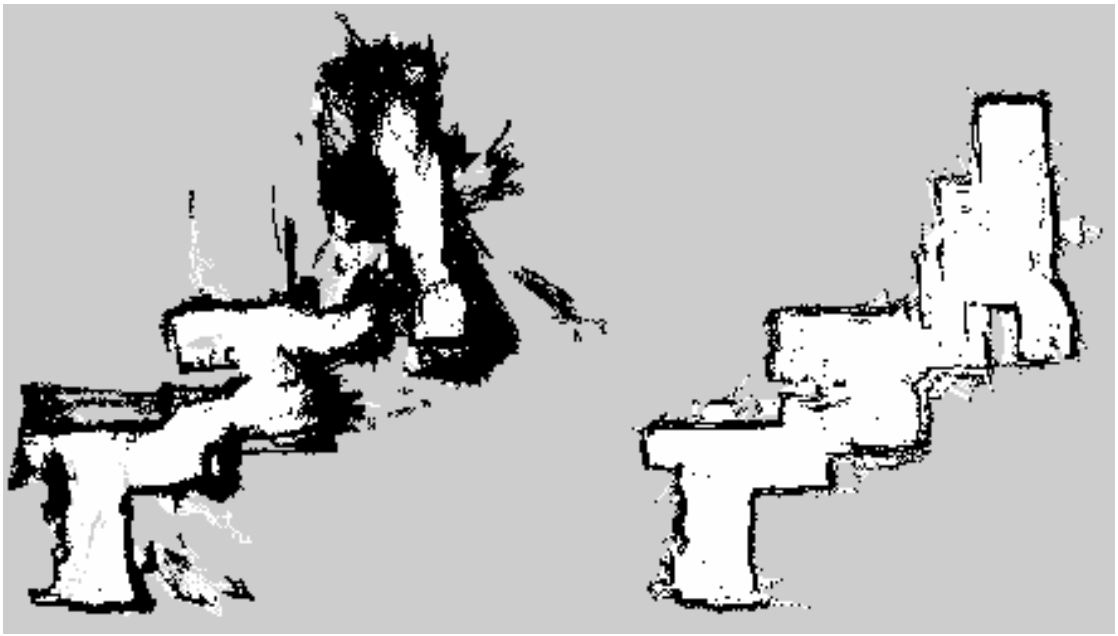
```
image: map.pgm
resolution: 0.050000
origin: [-100.000000, -100.000000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

Figure 6.2: Sample of yaml file created alongside the occupancy grid map, and reporting some useful information about it.

In regards of the created map, as said before, the use of RTAB-Map was abandoned for the results obtained on the occupancy grid map, which were too dirty to allow a proper work on theme. In Figure 6.3 the difference between the result obtained with RTAB-Map (to the left) and the one obtained using GMapping (to the right) is well evident. In particular, the latter is much more clear and clean, and allows a post processing operation to enhance it even more. However, how said before, it is worth noting that, contrary to the one obtained with GMapping, the RTAB-Map map has been obtained only with the use of the two RealSense cameras, without the lidar.



(a)



(b)

Figure 6.3: Comparison between (a) reconstruction of the real map of the environment used to test the mapping algorithm and (b) the two Occupancy Grid Map obtained. To the left the one obtained with RTAB-Map, using only the two cameras. To the right the one obtained with GMapping, using the lidar alongside RealSense T265.

6.2 Prediction of environment frames

At the end of the slam and acquisition operation, `Predict.launch` can be executed. The first node it run, `X_Y_Pred_generator.py`, retrieve the frames from the directory *acquisition*, preprocess them in order to give to each frame the same format of images used for training, and then use the previously trained model to make a prediction.

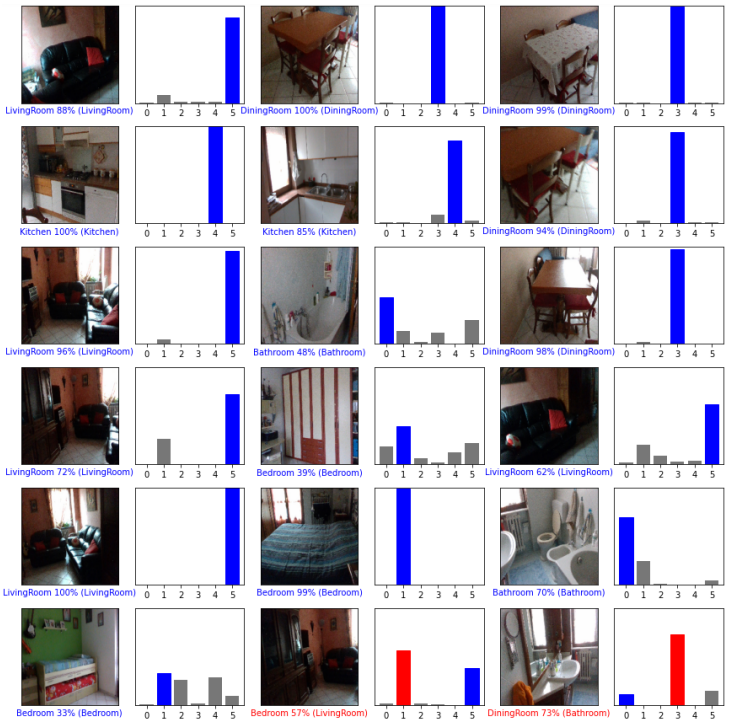
Here, one of the major problems of this project arises. The training set used on the model is a collection of photography of domestic rooms taken from a man-height, and the set used to test the accuracy of such model contains similar images. Testing this model on the field, if the Intel RealSense D435i is maintained to a man-height, the results are pretty accurate, consistent with the accuracy values found with the validation set. On the other hand, during the navigation of the robot, frames are acquired at a ground-height, and this inevitably influence negatively the accuracy of the model, how can be seen in Figure 6.5.

Although this problem is obvious, it can indeed be solved rather easily. In fact, since the model itself proved to be very accurate and robust, a simple retrain with a dataset of images taken from a ground-height should be sufficient to obtain vastly better results.

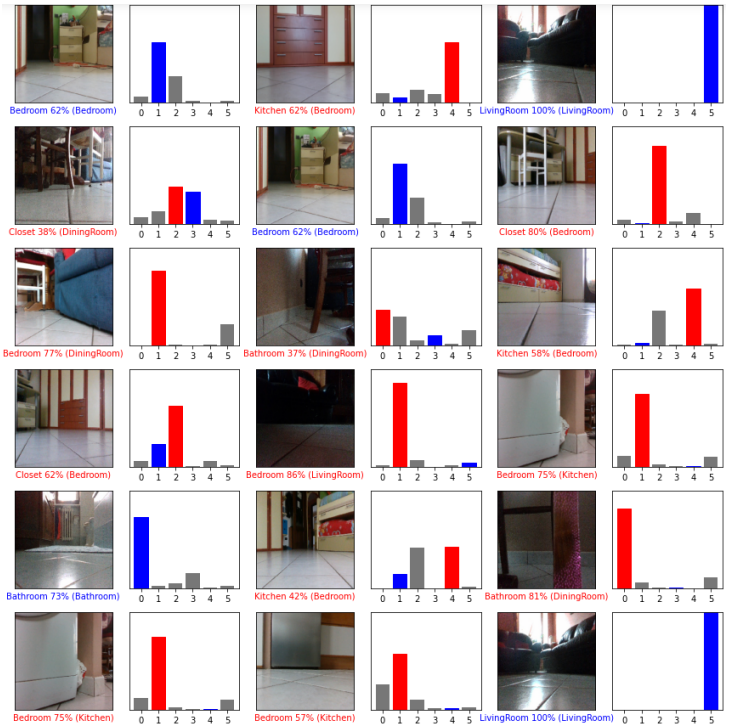
Once all the frames are been predicted, `X_Y_Pred_generator.py` retrieve the data contained in `robot_pose.csv` and add prediction information to it. Then this new vector containing poses and predictions undergoes a mean operation, after which for every n sample, where n is equal to ten by default but can be selected by the user, only one sample remains, which posses as pose value the mean of the n poses, and as prediction the mode of the n predictions. The remaining vector is then saved in a new CSV file, called `X_Y_Pred.csv`, similar to that exposed in Figure 6.4.

3.13E-01	-3.20E-01	1.00E+00
4.50E-01	-1.20E+00	1.00E+00
7.77E-01	6.23E-01	1.00E+00
3.79E+00	1.50E+00	0.00E+00
3.05E+00	2.62E+00	1.00E+00
4.68E+00	2.52E+00	0.00E+00
5.94E+00	3.94E+00	0.00E+00
7.05E+00	3.48E+00	1.00E+00
6.45E+00	6.37E+00	1.00E+00
5.90E+00	6.32E+00	1.00E+00

Figure 6.4: Sample of CSV file created by `X_Y_Pred_generator.py`. The first column corresponds to x coordinates, the second to y coordinates, and the third to the predictions, where each number correspond to a different class.



(a)



(b)

Figure 6.5: Comparison of the accuracy of the model on (a) man-height frames and (b) ground-height frames.

6.3 Map Post-Processing

The execution of the second node, `Mark_map.py`, is delayed since it needs `X_Y_Pred.csv` to work, and so has to wait for `X_Y_Pred_generator.py` to finish. First of all, this node apply a post-process operation on the raw map, `map.pgm`. After the application of a threshold, a gaussian blur and a Canny filter, with some further arrangement, the final map can be observed in Figure 6.6. How can be seen, the map is now much cleaner, most of biased pixels are removed, and only the real obstacles and walls are maintained.

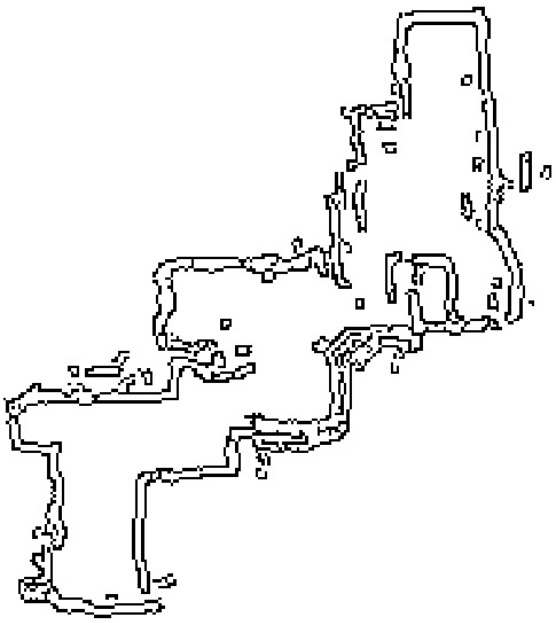


Figure 6.6: Post processed image of the Map.

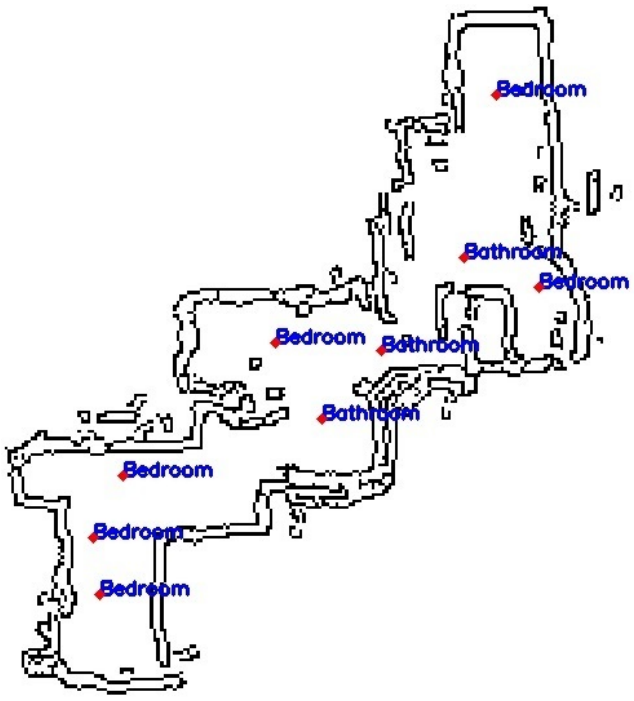
On the other hand, this result clearly is not as accurate as expected initially, and here arises the second big problem of this project. While the major walls and obstacles are correctly detected, they are not always well defined, and lots of imperfections still affect the final outcome. These errors can have principally two causes:

- The last data (and so the final results), have been gathered in unfavorable or, in any case, not optimal conditions. While in the early testing the control of the robot was actually actuated through teleoperations, for late testing the robot was driven manually. This, inevitably, has certainly negatively influenced the data acquired, and consequently the map obtained. Driving manually the robot instead of controlling it through teleoperations, and so letting it move on its own wheels, cause not very predictable and clean movements of the robot and so of its sensors. Furthermore, since the SLAM algorithm is

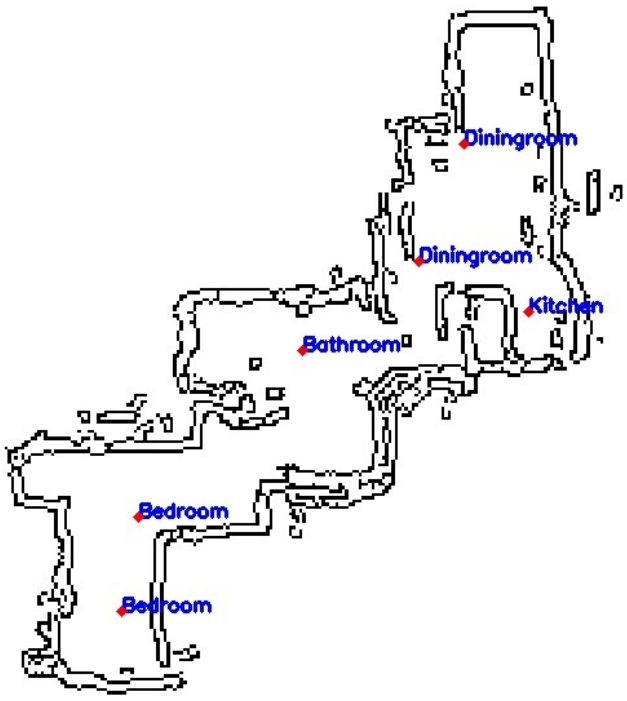
mainly designed for mapping environments with no mobile subjects, the presence of close quarter operator to push it has caused inevitably a not indifferent element of disturbance;

- While Intel RealSense cameras are commercial sensors, but still surprisingly precise and efficient in their work, the lidar unit mounted on the TurtleBot3 Waffle is very elemental and not so accurate. The employment of a better sensor of this kind could certainly highly improve results.

Now, the map is enlarged in order to allow the insertion of labels. Then two coordinate conversions are carried out. The first is used for convert from real world coordinates to image coordinates, using the information coming from `map.yaml`. The second is used for convert from the original image coordinates to the enlarged image coordinates. With these two transformations, it is possible to locate the poses contained in `X_Y_Pred.csv` on the map image. A further function is used to convert the numeric predictions contained in the CSV files to text and, knowing the poses, these textual predictions are marked on the map as labels. The final result can be observed in Figure 6.7(a), while (b) represent the same result using frame images taken from man-height, for comparison.



(a)



(b)

Figure 6.7: Post processed images of the Map with labels.

Chapter 7

Conclusions

Results obtained can be considered acceptable, satisfy the request of the thesis, and represent a first step of a larger project. The author of this project is aware of problems still afflicting the outcomes, and discussed in the previous chapter, so here some implementations which can reveal useful to improve results are presented:

- How said before, the Room Recognition algorithm is greatly influenced by the train set used for the learning. Since training the model with man-height taken photographs and testing it on the field with similar data has provided satisfying outcomes, a retrain with a ground-height dataset could certainly improve predictions;
- Retest the SLAM algorithm in better conditions, and possibly with a more accurate lidar sensor can also vastly improve the mapping results.

On the other hand, this project opens up to a series of future implementations:

- The algorithm can be further improved by adding an application able to segment rooms using only the occupancy grid map, so it can place a single label for each room, which is the mean value of all the predictions taken in that room;
- It can be integrated with navigation and object avoidance algorithms, to allow autonomous operations within the environment;
- It can be integrated with additional hardware components such as grappling end effectors, to allow assistance to disabled or elderly users;

Furthermore, it is worth noting that, from a user privacy perspective, all the images and information gathered are fully processed right on the robot, and are not shared, by any means, with external entities. This makes it optimal for working in any private environment.

Bibliography

[1] T. Mitchell. *Machine Learning*. McGraw Hill, 1997. ISBN: 978-0-07-042807-2.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[3] Aurélien Géron. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*. 2nd edition. O'Reilly Media, Inc., 2019. ISBN: 9781492032649.

[4] Oliver Kramer. *Dimensionality Reduction with Unsupervised Nearest Neighbors*. Vol. 51. June 2013. ISBN: 978-3-642-38652-7. DOI: 10.1007/978-3-642-38652-7.

[5] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[6] Sciavicco L. Villani L. Oriolo G Siciliano B. *Robotics Modelling, Planning and Control*. Springer, 2009. ISBN: 978-1-84628-642-1.

[7] Thomas Bräunl. *Embedded Roboticsl*. Springer, 2006. ISBN: 978-3-540-34319-6.

[8] Gerkey B. Smart W. D. Quigley M. *Programming Robots with ROS*. O'Reilly Media, Inc., 2015. ISBN: 9781449323899.

[9] ROBOTIS turtlebot3 series e manual. https://emanual.robotis.com/docs/en/platform/turtlebot3/specifications/.

[10] ROBOTIS official website. https://www.robotis.com/.

[11] NVIDIA Jetson AGX Xavier. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/.

[12] NVIDIA JetPack SDK. https://developer.nvidia.com/embedded/jetpack.

[13] Software R&D Manager at Intel Corporation on GitHub Sergey Dorodnicov. https://github.com/IntelRealSense/librealsense/issues/3642.

[14] Intel Realsense D435i Depth Camera. https://www.intelrealsense.com/depth-camera-d435i/.

[15] Intel Realsense T265 Tracking Camera. https://www.intelrealsense.com/tracking-camera-t265/.

[16] Intel Realsense T265 and D400 series integration. https://www.intelrealsense.com/depth-and-tracking-combined-get-started/.

[17] Lidar module at ROBOTIS official website. https://emanual.robotis.com/docs/en/platform/turtlebot3/appendix_lds_01/.

[18] Burgard Wolfram Fox Dieter Thrun Sebastian. *Probabilistic Robotics*. The MIT Press, 2005. ISBN: 978-0-262-20162-9.

[19] Gaussian Distribution Wikimedia Common. https://commons.wikimedia.org/wiki/Category:Normal_distribution#/media/File:Gaussian_distribution.svg.

[20] Cyrill Stachniss. *Robot Mapping, Least Squares*. lecture slides. http://ais.informatik.uni-freiburg.de/teaching/ws13/mapping/pdf/slam14-least-squares.pdf. Albert-Ludwigs-Universitat Freiburg, 2014.

[21] Cyrill Stachniss. *Robot Mapping, Least Squares Approach to SLAM*. lecture slides. http://ais.informatik.uni-freiburg.de/teaching/ws13/mapping/pdf/slam15-ls-slam.pdf. Albert-Ludwigs-Universitat Freiburg, 2014.

[22] Paul Newman Kin Leong Ho. "Loop closure detection in SLAM by combining visual and spatial appearance". In: *Robotics and Autonomous Systems* 54.9 (2006). Selected papers from the 2nd European Conference on Mobile Robots (ECMR '05), pp. 740 –749. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2006.04.016. URL: http://www.sciencedirect.com/science/article/pii/S0921889006000844.

[23] RTAB-Map official website. http://introlab.github.io/rtabmap/.

[24] Scikit learn official website. https://scikit-learn.org/stable/.

[25] TensorFlow official website. https://www.tensorflow.org/.

[26] Scikit learn confusion matrix. https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html.

[27] Ramprasaath R. Selvaraju et al. "Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization". In: *International Journal of Computer Vision* 128.2 (2019). ISSN: 1573-1405. DOI: 10.1007/s11263-019-01228-7. URL: \url{http://dx.doi.org/10.1007/s11263-019-01228-7}.

[28] A. Quattoni and A. Torralba. "Recognizing indoor scenes". In: (2009, pages=).

[29] Shaoqing Ren Jian Sun Kaiming He Xiangyu Zhang. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: \url{http://arxiv.org/abs/1512.03385}.

[30] Bo Chen Dmitry Kalenichenko Weijun Wang Tobias Weyand Marco Andreetto Hartwig Adam Andrew G. Howard Menglong Zhu. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: *CoRR* abs/1704.04861 (2017). arXiv: 1704.04861. URL: \url{http://arxiv.org/abs/1704.04861}.

[31] Antoni Rosinol et al. "Kimera: an Open-Source Library for Real-Time Metric-Semantic Localization and Mapping". In: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*. 2020. URL: \url{https://github.com/MIT-SPARK/Kimera}.

[32] Intel Realsense ROS wrapper at GitHub. https://github.com/IntelRealSense/realsense-ros.

[33] Madgwick IMU filter at GitHub. https://github.com/ccny-ros-pkg/imu_tools.

[34] Robot Localization Plugin at GitHub. https://github.com/cra-ros-pkg/robot_localization.

[35] Cyrill Stachniss Wolfram Burgard Gmapping at OpenSLAM Giorgio Grisetti. https://openslam-org.github.io/gmapping.html.

[36] OpenCV official website. https://opencv.org/.

[37] CvBridge at ROS official site. http://wiki.ros.org/cv_bridge.